Delhi University (DU)
B.Sc. (H) Computer Science
Sem - IV
Design and Analysis of Algorithms

Visit https://www.tutorialsduniya.com for Compiled books, Notes, books, programs, question papers with solutions etc.

Facebook: https://www.facebook.com/tutorialsduniya
Youtube: https://www.youtube.com/user/TutorialsDuniya
LinkedIn: https://www.linkedin.com/company/tutorialsduniya
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
## Guidelines: Design and Analysis of Algorithms

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Topic</th>
<th>Reference</th>
<th>Total Hours</th>
<th>Weightage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>Divide and Conquer Techniques</td>
<td>4.3-4.6 [1]</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>Linear time Sorting Algorithms, Medians and Order Statistics</td>
<td>Ch. 8 [2]</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.1,9.2 (Without Analysis) [2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Graph Algorithms</td>
<td>Ch. 3 [3]</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Amortized Analysis</td>
<td>17.1 [2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Greedy Technique</td>
<td>4.1, 4.2, 4.4, 4.5 (excluding reverse delete algorithm), 4.6 [3]</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

**References:**


[3]. Algorithm Design, Kleinberg and Tardos, Pearson Publication

Visit [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com) for Notes, books, programs, question papers with solutions etc.

For any query, contact us at help@tutorialsduniya.com
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
Lab Assignments List:

1. Implement Bubble, selection, insertion, merge, quick sort. And count the number of comparisons in each case.
2. Implement Heap Sort (The program should report the number of comparisons)
3. Implement Radix Sort
4. Write a program to Implement RB Tree supporting following operations:
   a. Insert a node
   b. Delete a node
   c. Search a number and report the color of node having this number
5. WAP to implement BFS in a graph represented via adjacency list.
6. WAP to implement DFS in a graph represented via adjacency list.
7. Using any greedy approach find the Minimum Spanning Tree of a graph.
8. Given a set of positive integers and a sum value S, find out if there exists a subset in array whose sum is equal to given sum S using Dynamic Programming.

For the algorithms at S.No 1 to 2, test run the algorithm on 100 different inputs of sizes varying from 30 to 1000. Count the number of comparisons and draw the graph. Compare it with a graphs of nlogn, n and n².

Note: Solutions of all these programs are available on https://www.tutorialsduniya.com

For any query, contact us at help@tutorialsduniya.com
TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
Computer Algorithms
Introduction to Design and Analysis
THIRD EDITION

Sara Baase
San Diego State University

Allen Van Gelder
University of California at Santa Cruz

For any copyright query, contact us at copyright@tutorialsduniya.com

ADDISON-WESLEY
An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Menlo Park, California • New York
Harlow, England • Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

For any query, contact us at help@tutorialsduniya.com
Computer Algorithms - Index

4 Sorting

4.1 Introduction 150
4.2 Insertion Sort 157
4.3 Divide and Conquer 159
4.4 Quicksort 174
4.5 Merging Sorted Sequences 171
4.6 Mergesort 174

Exercises 206
Programs 221

11 String Matching

11.1 Introduction 484
11.2 A Straightforward Solution 485
11.3 The Knuth-Morris-Pratt Algorithm 487

Exercises 508
Programs 512

For any copyright query, contact us at copyright@tutorialsduniya.com

For any query, contact us at help@tutorialsduniya.com
# Introduction to Algorithms - Index

## 8 Sorting in Linear Time 191
- 8.1 Lower bounds for sorting 191
- 8.2 Counting sort 194
- 8.3 Radix sort 197
- 8.4 Bucket sort 200

## 9 Medians and Order Statistics 213
- 9.1 Minimum and maximum 214
- 9.2 Selection in expected linear time 215

## 13 Red-Black Trees 308
- 13.1 Properties of red-black trees 308
- 13.2 Rotations 312
- 13.3 Insertion 315
- 13.4 Deletion 323

## 17 Amortized Analysis 451
- 17.1 Aggregate analysis 452

For any copyright query, contact us at copyright@tutorialsduniya.com
Algorithm Design

JON KLEINBERG • ÉVA TARDOS
Cornell University

For any copyright query, contact us at copyright@tutorialsduniya.com
# Algorithm Design - Index

## 3 Graphs

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Basic Definitions and Applications</td>
<td>73</td>
</tr>
<tr>
<td>3.2</td>
<td>Graph Connectivity and Graph Traversal</td>
<td>78</td>
</tr>
<tr>
<td>3.3</td>
<td>Implementing Graph Traversal Using Queues and Stacks</td>
<td>87</td>
</tr>
<tr>
<td>3.4</td>
<td>Testing Bipartiteness: An Application of Breadth-First Search</td>
<td>94</td>
</tr>
<tr>
<td>3.5</td>
<td>Connectivity in Directed Graphs</td>
<td>97</td>
</tr>
<tr>
<td>3.6</td>
<td>Directed Acyclic Graphs and Topological Ordering</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>Solved Exercises</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>107</td>
</tr>
</tbody>
</table>

## 4 Greedy Algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Interval Scheduling: The Greedy Algorithm Stays Ahead</td>
<td>116</td>
</tr>
<tr>
<td>4.2</td>
<td>Scheduling to Minimize Lateness: An Exchange Argument</td>
<td>125</td>
</tr>
<tr>
<td>4.4</td>
<td>Shortest Paths in a Graph</td>
<td>137</td>
</tr>
<tr>
<td>4.5</td>
<td>The Minimum Spanning Tree Problem</td>
<td>142</td>
</tr>
<tr>
<td>4.6</td>
<td>Implementing Kruskal’s Algorithm: The Union-Find Data Structure</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>Solved Exercises</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>188</td>
</tr>
</tbody>
</table>

## 6 Dynamic Programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Weighted Interval Scheduling: A Recursive Procedure</td>
<td>252</td>
</tr>
<tr>
<td>6.2</td>
<td>Principles of Dynamic Programming: Memoization or Iteration over Subproblems</td>
<td>258</td>
</tr>
<tr>
<td>6.4</td>
<td>Subset Sums and Knapsacks: Adding a Variable</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td>Solved Exercises</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>312</td>
</tr>
</tbody>
</table>

For any copyright query, contact us at copyright@tutorialsduniya.com

For any query, contact us at help@tutorialsduniya.com
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Iterative Techniques</td>
<td>Ch 4.1, 4.2 [1] Ch 6 [2]</td>
</tr>
</tbody>
</table>

**TutorialsDuniya.com**

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
Sorting

* 4.1  Introduction
4.2  Insertion Sort
4.3  Divide and Conquer
4.4  Quicksort
4.5  Merging Sorted Sequences
4.6  Mergesort
4.7  Lower Bounds for Sorting by Comparison of Keys
4.8  Heapsort
4.9  Comparison of Four Sorting Algorithms
4.10 Shellsort
4.11 Radix Sorting

For any query, contact us at help@tutorialsduniya.com
Chapter 4  Sorting

4.1 Introduction

In this chapter we will study several algorithms for sorting, that is, for arranging the elements of a set into order. The problem of sorting a set of objects was one of the first intensely studied computer science problems. Many of the best-known applications of the Divide-and-Conquer algorithm design paradigm are sorting algorithms. During the 1960s, when commercial data processing became automated on a large scale, the sort program was the most frequently run program at many computer installations. One software company stayed in business for years on the strength of its better sort program. With today’s hardware, the performance issues of sorting have shifted somewhat. In the 1960s, transferring data between slow storage (tape or disk) and main memory was a major performance bottleneck. Main memory was in the neighborhood of 100,000 bytes, and files to be sorted were orders of magnitude larger. Algorithms to perform this kind of sorting were the main focus of attention. Today, main memories 1,000 times that size (i.e., 100 megabytes) are commonplace, and 10,000 times that size (a few gigabytes) are available. So most files can fit in main memory.

There are several good reasons for studying sorting algorithms. First, they are of practical use because sorting is done often. Just as having the entries in telephone books and dictionaries in alphabetical order makes them easy to use, working with large sets of data in computers is facilitated when the data are sorted. Second, quite a lot of sorting algorithms have been devised (more than will be covered here), and studying a number of them should impress upon you the fact that you can take many different points of view toward the same problem. The discussion of the algorithms in this chapter should provide some insights on the questions of how to improve a given algorithm and how to choose among several. Third, sorting is one of few problems for which we can easily derive strong lower bounds for worst case and average behavior. The bounds are strong in the sense that there are algorithms that do approximately the minimum amount of work specified. Thus we have essentially optimal sorting algorithms.

In the descriptions of most of the algorithms, we assume the set to be sorted is stored as an array, so that the element at any position can be accessed at any time; this is called random access. However, some of the algorithms are useful for sorting files and linked lists, as well. When the set is only accessible in a sequential fashion, we use the term sequence, to emphasize that the structure might be a linked list or sequential file, as well as an array. If an array is defined over the range of indexes $0, \ldots, n-1$, then a range or subrange of that array is a contiguous sequence of entries between two specified indexes, $first$ and $last$, such that $0 \leq first$ and $last \leq n-1$. If $last < first$, the range is said to be empty.

We assume that each element in the set to be sorted contains an identifier, called a key, which is an element of some linearly ordered set, and that two keys can be compared to determine which is larger or that they are equal. We always sort keys into nondecreasing order. Each element in the set might contain other information aside from the key. When keys are rearranged during the sorting process, the associated information is also rearranged as appropriate, but sometimes we refer only to the keys and make no explicit mention of the rest of the entry.
The algorithms considered in Sections 4.2 through 4.10 are all from the class of sorting algorithms that may compare keys (and copy them) but must not do other operations on the keys. We call these “algorithms that sort by comparison of keys,” or “comparison-based algorithms,” for short. The measure of work primarily used for analyzing algorithms in this class is the number of comparisons of keys. In Section 4.7, lower bounds on the number of comparisons performed by such algorithms are established. Section 4.11 discusses sorting algorithms for which operations other than comparisons of keys are available, and for which different measures of work are appropriate.

The algorithms in this chapter are called internal sorts because the data are assumed to be in the computer’s high-speed, random-access memory. Different performance issues arise for sorting data sets that are too large to fit in memory. Algorithms for sorting large sets of data stored on external, slower storage devices with restrictions on the way data are accessed are called external sorts. See Notes and References at the end of the chapter for sources on such algorithms.

When analyzing sorting algorithms, we will consider how much extra space they use (in addition to the input). If the amount of extra space is constant with respect to the input size, the algorithm is said to work in place.

To help make the algorithms as clear as possible, we use Element and Key as type identifiers, but treat Key as a numeric type in that we use the relational operators “=, !=, <,” and so on. When the book has a key-comparison expression like “d[i].key < x,” if the actual types are nonnumeric (String, for example), a Java program requires syntax involving a method call, such as “less(d[i].key, x).” This holds for many languages besides Java.

Java sidenote: By means of the Comparable interface in Java, it is possible to write one procedure that is able to compare a wide variety of key types. The type name Key would be replaced by the key word Comparable. Some details are given in Appendix A. Recall that an array with entries of type Element is declared as

```
Element[] arrayName;
```

in Java.

4.2 Insertion Sort

Insertion Sort is a good sorting algorithm to begin with because the idea behind it is a natural and general one, and its worst-case and average-behavior analyses are easy. It is also used as part of a faster sorting algorithm that we describe in Section 4.10.

4.2.1 The Strategy

We begin with a sequence $E$ of $n$ elements in arbitrary order, as illustrated by Figure 4.1. (Insertion Sort can be used on keys from any linearly ordered set, but for the stick figure illustrations, think of the keys as the heights of the sticks, which are the elements.)

Suppose we have sorted some initial segment of the sequence. Figure 4.2 shows a snapshot of the sequence after the five elements on the left end have been sorted. The
general step is to increase the length of the sorted segment by inserting the next element in its proper place.

Let $x$ be the next element to be inserted in the sorted segment, that is, $x$ is the leftmost element in the unexamined segment. First we pull $x$ "out of the way" (that is, copy it to a local variable), leaving a vacancy in its former position. Then we repeatedly compare $x$ to the element just to the left of the vacancy, and as long as $x$ is smaller, we move the that element into the vacancy, thereby leaving a vacancy where it was. That is, the vacancy shifts one place to the left. This process stops when we run out of elements to the left of the current vacancy, or when the element to the left of the current vacancy is smaller than or equal to $x$. Then $x$ is inserted in the vacancy, as shown in Figure 4.3. To get the algorithm started, we need only observe that the first element alone can be considered a sorted segment. As we formalize this into a procedure, we assume the sequence is an array; however, the idea works with lists and other sequential structures.
4.2 Insertion Sort

**int shiftVac(Element[*] E, int vacant, Key x)\**

*Precondition:* vacant is nonnegative.

*Postconditions:* Let xLoc be the value returned to the caller. Then:

1. Elements in \(E\) at indexes less than xLoc are in their original positions and have keys less than or equal to \(x\).
2. Elements in \(E\) at positions xLoc + 1, ..., vacant are greater than \(x\) and were shifted up by one position from their positions when shiftVac was invoked.

![Figure 4.4](https://www.tutorialsduniya.com) Specifications for shiftVac

### 4.2.2 The Algorithm and Analysis

We now spell out the sorting procedure in more detail. Let the subroutine shiftVac(\(E\), vacant, \(x\)) have the job of shifting elements until the vacancy is at the correct position in which to place \(x\) among the sorted elements. The procedure returns the index of the vacancy, say xLoc, to the caller. The preconditions and postconditions are stated in Figure 4.4. In other words, shiftVac makes the transition from Figure 4.2 to Figure 4.3. Now insertionSort can just keep calling shiftVac, making a longer and longer sorted segment at the left end, until all elements are in the sorted segment.

The shiftVac procedure takes a typical form for generalized searching routines (Definition 1.12). If there is no more data to look at, fail; else look at one data item, and if it is what we are looking for, succeed; otherwise continue with the unexamined data. Because there are two terminating cases, this can be awkward for a while loop, unless a break is used for one or more of the terminating cases. The recursive formulation is straightforward.

**int shiftVacRec(Element[*] E, int vacant, Key x)\**

1. if (vacant == 0)
2. xLoc = vacant;
3. else if (E[vacant-1].key <= x)
4. xLoc = vacant;
5. else
6. E[vacant] = E[vacant-1];
7. xLoc = shiftVacRec(E, vacant-1, x);
8. return xLoc;

To verify that we are using recursion properly on line 7, we note that the recursive call is working on a smaller range, and its second argument is nonnegative, so the precondition (stated in Figure 4.4) is satisfied. (You should check the chain of reasoning for why vacant - 1 is nonnegative—why can't it be negative?) Correctness is now straightforward if we remember that we can *assume* that the recursive call on line 7 accomplishes its objective.

For any query, contact us at help@tutorialsduniya.com
Chapter 4  Sorting

Although the procedure for shiftVacRec is very simple, if we visualize the activation trace for the ith element of E to be inserted, we realize that the depth of recursion, or the frame stack, could grow to size n. This could be undesirable for large n. Therefore this is a case where the recursion should be changed into an iteration, after we are sure everything is working correctly. (Trying to optimize a nonworking program is surely an exercise in futility.) The purpose is not so much to save time as to conserve space. Actually, many compilers, if told to optimize shiftVacRec, will perform this transformation automatically. The full algorithm below includes the iteratively coded version of shiftVac.

Algorithm 4.1  Insertion Sort

Input: E, an array of elements, and n ≥ 0, the number of elements. The range of indexes is 0, . . . , n − 1.

Output: E, with elements in nondecreasing order of their keys.

Remark: The specifications for the shiftVac subroutine are given in Figure 4.4.

void insertionSort(Element[] E, int n)
int xindex;
for (xindex = 1; xindex < n; xindex++)
    Element current = E[xindex];
    Key x = current.key;
    int xLoc = shiftVac(E, xindex, x);
    E[xLoc] = current;
return;

int shiftVac(Element[] E, int xindex, Key x)
int vacant, xLoc;
vacant = xindex;
xLoc = 0;  // Assume failure.
while (vacant > 0)
    if (E[vacant−1].key ≤ x)
        xLoc = vacant;  // Succeed.
        break;
    E[vacant] = E[vacant−1];
    vacant --;  // Keep looking.
return xLoc;

Worst-Case Complexity

For the analysis, we use i for xindex. For each value of i, the maximum number of key comparisons possible (in one call to the iterative shiftVac, or one top-level call of the recursive shiftVacRec) is i. Thus the total is
4.2 Insertion Sort  155

\[ W(n) \leq \sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}. \]

Note that we have established an upper bound on the worst-case behavior; it takes a moment of thought to verify that there are indeed inputs for which \( n(n - 1)/2 \) comparisons are done. One such worst case is when the keys are in reverse (i.e., decreasing) order. So

\[ W(n) = \frac{n(n - 1)}{2} \in \Theta(n^2). \]

**Average Behavior**

We assume that all permutations of the keys are equally likely as input. We will first determine how many key comparisons are done on the average to insert one new element into the sorted segment, that is, one call of shiftVac, for any particular value of \( i \) (used for xIndex). To simplify the analysis, we assume that the keys are distinct. (The analysis is very similar to that done for the Sequential Search algorithm in Chapter 1.)

There are \( i + 1 \) positions where \( x \) may go. Figure 4.5 shows how many comparisons are done depending on the position.

The probability that \( x \) belongs in any one specific position is \( 1/(i + 1) \). (This depends on the fact that \( x \) has not been examined earlier by the algorithm. If the algorithm had made any earlier decisions based on the value of \( x \), we could not necessarily assume that \( x \) is uniformly random with respect to the first \( i \) keys.) Thus the average number of comparisons in shiftVac to find the location for the \( i \)th element is

\[
\frac{1}{i+1} \sum_{j=1}^{i} j + \frac{1}{i+1} = \frac{i}{2} + \frac{1}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}.
\]

Now, adding for all \( n - 1 \) insertions,

\[
A(n) = \sum_{i=1}^{n-1} \left( \frac{i}{2} + 1 - \frac{1}{i+1} \right) = \frac{n(n - 1)}{4} + n - 1 - \frac{1}{j}.
\]

For any query, contact us at help@tutorialsduniya.com
where we substituted \( j = i + 1 \) to get the last sum. We saw from Equation (1.16) that
\[
\sum_{j=1}^{n}(1/j) \approx \ln n,
\]
and we can incorporate the 1 preceding the sum to make the lower limit \( j = 1 \). Ignoring lower-order terms, we have
\[
A(n) \approx \frac{n^2}{4} \in \Theta(n^2).
\]

**Space**

Clearly, Insertion Sort is an in-place sort with the iterative version of shiftVac. With the recursive version the frame stack can grow to \( \Theta(n) \).

### 4.2.3 Lower Bounds on the Behavior of Certain Sorting Algorithms

Think of the element whose key is \( x \) as occupying the "vacant" position in the array while Insertion Sort compares \( x \) to the key to its left. Then after each comparison, Insertion Sort either moves no elements or simply interchanges two adjacent elements. We will show that all sorting algorithms that do such limited, "local" moving of elements after each comparison must do about the same amount of work as Insertion Sort.

A permutation on \( n \) elements can be described by a one-to-one function from the set \( N = \{1, 2, \ldots, n\} \) onto itself. There are \( n! \) distinct permutations on \( n \) elements. Let the elements in the unsorted sequence \( E \) be \( x_1, x_2, \ldots, x_n \). To simplify the notation in this discussion, let's assume that the elements to be sorted are stored in positions \( 1, 2, \ldots, n \) of \( E \), rather than in \( 0, 1, \ldots, n - 1 \). There is a permutation \( \pi \) such that, for \( 1 \leq i \leq n \), \( \pi(i) \) is the correct position of \( x_i \) when the sequence is sorted. Without loss of generality, we can assume that the keys are the integers \( 1, 2, \ldots, n \) since we can substitute 1 for the smallest key, 2 for the next smallest, and so on, without causing any changes in the instructions carried out by the algorithm. Then the unsorted input is \( \pi(1), \pi(2), \ldots, \pi(n) \). For example, consider the input sequence \( 2, 4, 1, 5, 3 \). \( \pi(1) = 2 \) means that the first key, 2, belongs in the second position, which it clearly does. \( \pi(2) = 4 \) because the second key, 4, belongs in the fourth position, and so on. We will identify the permutation \( \pi \) with the sequence \( \pi(1), \pi(2), \ldots, \pi(n) \).

An inversion of the permutation \( \pi \) is a pair \((\pi(i), \pi(j))\) such that \( i < j \) and \( \pi(i) > \pi(j) \). If \((\pi(i), \pi(j))\) is an inversion, the \( i \)th and \( j \)th keys in the sequence are out of order relative to each other. For example, the permutation \( 2, 4, 1, 5, 3 \) has four inversions \((2, 1), (4, 1), (4, 3), \) and \((5, 3) \). If a sorting algorithm removes at most one inversion after each key comparison (say, by interchanging adjacent elements, as Insertion Sort does), then the number of comparisons performed on the input \( \pi(1), \pi(2), \ldots, \pi(n) \) is at least the number of inversions of \( \pi \). So we investigate inversions.

It is easy to show that there is a permutation that has \( n(n - 1)/2 \) inversions. (Which permutation?) Thus the worst-case behavior of any sorting algorithm that removes at most one inversion per key comparison must be in \( \Omega(n^2) \).

To get a lower bound on the average number of comparisons done by such sorting algorithms, we compute the average number of inversions in permutations. Each permutation \( \pi \) can be paired off with its transpose permutation \( \pi(n), \pi(n - 1), \ldots, \pi(1) \). For example,
the transpose of 2, 4, 1, 5, 3 is 3, 5, 1, 4, 2. Each permutation has a unique transpose and is distinct from its transpose (for \( n > 1 \)). Let \( i \) and \( j \) be integers between 1 and \( n \), and suppose \( j < i \). Then \((i, j)\) is an inversion in exactly one of the permutations \( \pi \) and transpose of \( n \). There are \( n(n - 1)/2 \) such pairs of integers. Hence each pair of permutations has \( n(n - 1)/2 \) inversions between them, and therefore an average of \( n(n - 1)/4 \). Thus, overall, the average number of inversions in a permutation is \( n(n - 1)/4 \), and we have proved the following theorem.

**Theorem 4.1** Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least \( n(n - 1)/2 \) comparisons in the worst case and at least \( n(n - 1)/4 \) comparisons on the average (for \( n \) elements).

Since Insertion Sort does \( n(n - 1)/2 \) key comparisons in the worst case and approximately \( n^2/4 \) on the average, it is about the best we can do with any algorithm that works "locally," for example, interchanging only adjacent elements. It is, of course, not obvious at this point that any other strategy can do better, but if there are significantly faster algorithms they must move elements more than one position at a time.
<table>
<thead>
<tr>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
</tr>
<tr>
<td>Programming in Java</td>
</tr>
<tr>
<td>Data Structures</td>
</tr>
<tr>
<td>Computer Networks</td>
</tr>
<tr>
<td>Android Programming</td>
</tr>
<tr>
<td>PHP Programming</td>
</tr>
<tr>
<td>JavaScript</td>
</tr>
<tr>
<td>Java Server Pages</td>
</tr>
<tr>
<td>Python</td>
</tr>
<tr>
<td>Microprocessor</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>Machine Learning</td>
</tr>
<tr>
<td>Computer System Architecture</td>
</tr>
<tr>
<td>Discrete Structures</td>
</tr>
<tr>
<td>Operating Systems</td>
</tr>
<tr>
<td>Algorithms</td>
</tr>
<tr>
<td>DataBase Management Systems</td>
</tr>
<tr>
<td>Software Engineering</td>
</tr>
<tr>
<td>Theory of Computation</td>
</tr>
<tr>
<td>Operational Research</td>
</tr>
<tr>
<td>System Programming</td>
</tr>
<tr>
<td>Data Mining</td>
</tr>
<tr>
<td>Computer Graphics</td>
</tr>
<tr>
<td>Data Science</td>
</tr>
</tbody>
</table>

- Programs: [https://www.tutorialsduniya.com/programs](https://www.tutorialsduniya.com/programs)
- Question Papers: [https://www.tutorialsduniya.com/question-papers](https://www.tutorialsduniya.com/question-papers)
- Python Notes: [https://www.tutorialsduniya.com/python](https://www.tutorialsduniya.com/python)
- Java Notes: [https://www.tutorialsduniya.com/java](https://www.tutorialsduniya.com/java)
- JavaScript Notes: [https://www.tutorialsduniya.com/javascript](https://www.tutorialsduniya.com/javascript)
- JSP Notes: [https://www.tutorialsduniya.com/jsp](https://www.tutorialsduniya.com/jsp)
- Microprocessor Notes: [https://www.tutorialsduniya.com/microprocessor](https://www.tutorialsduniya.com/microprocessor)
- OR Notes: [https://www.tutorialsduniya.com/operational-research](https://www.tutorialsduniya.com/operational-research)
6 Heapsort

In this chapter, we introduce another sorting algorithm: heapsort. Like merge sort, but unlike insertion sort, heapsort’s running time is $O(n \log n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a “heap,” to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

The term “heap” was originally coined in the context of heapsort, but it has since come to refer to “garbage-collected storage,” such as the programming languages Java and Lisp provide. Our heap data structure is not garbage-collected storage, and whenever we refer to heaps in this book, we shall mean a data structure rather than an aspect of garbage collection.

6.1 Heaps

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A$ that represents a heap is an object with two attributes: $A.length$, which (as usual) gives the number of elements in the array, and $A.heap-size$, which represents how many elements in the heap are stored within array $A$. That is, although $A[1..A.length]$ may contain numbers, only the elements in $A[1..A.heap-size]$, where $0 \leq A.heap-size \leq A.length$, are valid elements of the heap. The root of the tree is $A[1]$, and given the index $i$ of a node, we can easily compute the indices of its parent, left child, and right child:
Figure 6.1  A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

\textbf{PARENT}(i)
1 \textbf{return} \lfloor i/2 \rfloor

\textbf{LEFT}(i)
1 \textbf{return} 2i

\textbf{RIGHT}(i)
1 \textbf{return} 2i + 1

On most computers, the \textbf{LEFT} procedure can compute $2i$ in one instruction by simply shifting the binary representation of $i$ left by one bit position. Similarly, the \textbf{RIGHT} procedure can quickly compute $2i + 1$ by shifting the binary representation of $i$ left by one bit position and then adding in a 1 as the low-order bit. The \textbf{PARENT} procedure can compute $\lfloor i/2 \rfloor$ by shifting $i$ right one bit position. Good implementations of heapsort often implement these procedures as “macros” or “inline” procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a \textit{heap property}, the specifics of which depend on the kind of heap. In a \textit{max-heap}, the \textit{max-heap property} is that for every node $i$ other than the root,

$A[\text{PARENT}(i)] \geq A[i],$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains
values no larger than that contained at the node itself. A **min-heap** is organized in
the opposite way; the **min-heap property** is that for every node \( i \) other than the root,

\[
A[\text{PARENT}(i)] \leq A[i].
\]

The smallest element in a min-heap is at the root.

For the heapsort algorithm, we use max-heaps. Min-heaps commonly implement priority queues, which we discuss in Section 6.5. We shall be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term “heap.”

Viewing a heap as a tree, we define the **height** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of \( n \) elements is based on a complete binary tree, its height is \( \Theta(\lg n) \) (see Exercise 6.1-2). We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take \( O(\lg n) \) time. The remainder of this chapter presents some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The **MAX-HEAPIFY** procedure, which runs in \( O(\lg n) \) time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in \( O(n \lg n) \) time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in \( O(\lg n) \) time, allow the heap data structure to implement a priority queue.

**Exercises**

6.1-1
What are the minimum and maximum numbers of elements in a heap of height \( h \)?

6.1-2
Show that an \( n \)-element heap has height \( \lfloor \lg n \rfloor \).

6.1-3
Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.
6.1-4
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5
Is an array that is in sorted order a min-heap?

6.1-6
Is the array with values 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 a max-heap?

6.1-7
Show that, with the array representation for storing an $n$-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Maintaining the heap property

In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY. Its inputs are an array $A$ and an index $i$ into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT($i$) and RIGHT($i$) are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ “float down” in the max-heap so that the subtree rooted at index $i$ obeys the max-heap property.

\begin{verbatim}
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if $l \leq A$.heap-size and $A[l] > A[i]$
4      largest = l
5  else largest = i
6  if $r \leq A$.heap-size and $A[r] > A[largest]$
7      largest = r
8  if largest $\neq i$
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
\end{verbatim}

Figure 6.2 illustrates the action of MAX-HEAPIFY. At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ is determined, and its index is stored in $\text{largest}$. If $A[i]$ is largest, then the subtree rooted at node $i$ is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes node $i$ and its...
6.2 Maintaining the heap property

Figure 6.2 The action of MAX-HEAPIFY(A, 2), where A.heap-size = 10. (a) The initial configuration, with A[2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging A[2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY(A, 4) now has i = 4. After swapping A[4] with A[9], as shown in (c), node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.

children to satisfy the max-heap property. The node indexed by largest, however, now has the original value A[i], and thus the subtree rooted at largest might violate the max-heap property. Consequently, we call MAX-HEAPIFY recursively on that subtree.

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the Θ(1) time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children’s subtrees each have size at most 2n/3—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

\[ T(n) \leq T(2n/3) + \Theta(1) \]
The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is \(T(n) = O(\lg n)\). Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height \(h\) as \(O(h)\).

**Exercises**

6.2-1
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY \((A, 3)\) on the array \(A = \{27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\}\).

6.2-2
Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY \((A, i)\), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

6.2-3
What is the effect of calling MAX-HEAPIFY \((A, i)\) when the element \(A[i]\) is larger than its children?

6.2-4
What is the effect of calling MAX-HEAPIFY \((A, i)\) for \(i > A.\text{heap-size}/2\)?

6.2-5
The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-6
Show that the worst-case running time of MAX-HEAPIFY on a heap of size \(n\) is \(\Omega(\lg n)\). (Hint: For a heap with \(n\) nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

6.3 Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array \(A[1..n]\), where \(n = A.\text{length}\), into a max-heap. By Exercise 6.1-7, the elements in the subarray \(A[([n/2] + 1)..n]\) are all leaves of the tree, and so each is
6.3 Building a heap

a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD-MAX-HEAP(A)
1 A.heap-size = A.length
2 for i = ⌊A.length/2⌋ downto 1
3 MAX-HEAPIFY(A, i)

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the for loop of lines 2–3, each node \( i + 1 \), \( i + 2 \), \ldots, \( n \) is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, \( i = \lfloor n/2 \rfloor \). Each node \( \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n \) is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node \( i \) are numbered higher than \( i \). By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node \( i \) a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes \( i + 1, i + 2, \ldots, n \) are all roots of max-heaps. Decrementing \( i \) in the for loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, \( i = 0 \). By the loop invariant, each node 1, 2, \ldots, \( n \) is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs \( O(\lg n) \) time, and BUILD-MAX-HEAP makes \( O(n) \) such calls. Thus, the running time is \( O(n \lg n) \). This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an \( n \)-element heap has height \( \lfloor \lg n \rfloor \) (see Exercise 6.1-2) and at most \( \lceil n/2^{h+1} \rceil \) nodes of any height \( h \) (see Exercise 6.3-3).

The time required by MAX-HEAPIFY when called on a node of height \( h \) is \( O(h) \), and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by
Figure 6.3  The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP.  (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index \(i\) refers to node 5 before the call MAX-HEAPIFY(\(A, i\)).  (b) The data structure that results. The loop index \(i\) for the next iteration refers to node 4.  (c)–(e) Subsequent iterations of the for loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. (f) The max-heap after BUILD-MAX-HEAP finishes.
6.4 The heapsort algorithm

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array \( A[1..n] \), where \( n = A.length \). Since the maximum element of the array is stored at the root \( A[1] \), we can put it into its correct final position.
by exchanging it with \( A[n] \). If we now discard node \( n \) from the heap—and we can do so by simply decrementing \( A.heap\)-size—we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property. All we need to do to restore the max-heap property, however, is call \textsc{Max-Heapify}(\( A, 1 \)), which leaves a max-heap in \( A[1 \ldots n - 1] \). The heapsort algorithm then repeats this process for the max-heap of size \( n - 1 \) down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

**HEAPSORT**\( (A) \)

1. \textsc{Build-Max-Heap}(\( A \))
2. \textbf{for} \( i = A.length \) \textbf{down to} 2
4. \( A.heap\)-size = \( A.heap\)-size - 1
5. \textsc{Max-Heapify}(\( A, 1 \))

Figure 6.4 shows an example of the operation of \textsc{Heapsort} after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the \textbf{for} loop of lines 2–5 and after each iteration.

The \textsc{Heapsort} procedure takes time \( O(n \log n) \), since the call to \textsc{Build-Max-Heap} takes time \( O(n) \) and each of the \( n - 1 \) calls to \textsc{Max-Heapify} takes time \( O(\log n) \).

**Exercises**

6.4-1
Using Figure 6.4 as a model, illustrate the operation of \textsc{Heapsort} on the array \( A = (5, 13, 2, 25, 7, 17, 20, 8, 4) \).

6.4-2
Argue the correctness of \textsc{Heapsort} using the following loop invariant:

At the start of each iteration of the \textbf{for} loop of lines 2–5, the subarray \( A[1 \ldots i] \) is a max-heap containing the \( i \) smallest elements of \( A[1 \ldots n] \), and the subarray \( A[i + 1 \ldots n] \) contains the \( n - i \) largest elements of \( A[1 \ldots n] \), sorted.

6.4-3
What is the running time of \textsc{Heapsort} on an array \( A \) of length \( n \) that is already sorted in increasing order? What about decreasing order?

6.4-4
Show that the worst-case running time of \textsc{Heapsort} is \( \Omega(n \log n) \).
Figure 6.4 The operation of HEAPSORT. (a) The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. (b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of $i$ at that time. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array $A$. 

For any query, contact us at help@tutorialsduniya.com
6.4-5  ★
Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

6.5  Priority queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We will focus here on how to implement max-priority queues, which are in turn based on max-heaps; Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called a key. A max-priority queue supports the following operations:

**INSERT**($S$, $x$) inserts the element $x$ into the set $S$, which is equivalent to the operation $S = S \cup \{x\}$.

**MAXIMUM**($S$) returns the element of $S$ with the largest key.

**EXTRACT-MAX**($S$) removes and returns the element of $S$ with the largest key.

**INCREASE-KEY**($S$, $x$, $k$) increases the value of element $x$’s key to the new value $k$, which is assumed to be at least as large as $x$’s current key value.

Among their other applications, we can use max-priority queues to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling **EXTRACT-MAX**. The scheduler can add a new job to the queue at any time by calling **INSERT**.

Alternatively, a min-priority queue supports the operations **INSERT**, **MINIMUM**, **EXTRACT-MIN**, and **DECREASE-KEY**. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls **EXTRACT-MIN** at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling **INSERT**.

For any query, contact us at help@tutorialsduniya.com
6.5 Priority queues

We shall see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 23 and 24.

Not surprisingly, we can use a heap to implement a priority queue. In a given application, such as job scheduling or event-driven simulation, elements of a priority queue correspond to objects in the application. We often need to determine which application object corresponds to a given priority-queue element, and vice versa. When we use a heap to implement a priority queue, therefore, we often need to store a handle to the corresponding application object in each heap element. The exact makeup of the handle (such as a pointer or an integer) depends on the application. Similarly, we need to store a handle to the corresponding heap element in each application object. Here, the handle would typically be an array index. Because heap elements change locations within the array during heap operations, an actual implementation, upon relocating a heap element, would also have to update the array index in the corresponding application object. Because the details of accessing application objects depend heavily on the application and its implementation, we shall not pursue them here, other than noting that in practice, these handles do need to be correctly maintained.

Now we discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM implements the MAXIMUM operation in $\Theta(1)$ time.

**HEAP-MAXIMUM**

1. \textbf{return} $A[1]$

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the for loop body (lines 3–5) of the HEAPSORT procedure.

**HEAP-EXTRACT-MAX**

1. \textbf{if} $A$.heap-size $< 1$
2. \hspace{1em} \textbf{error} “heap underflow”
3. \hspace{1em} $max = A[1]$
5. \hspace{1em} $A$.heap-size $= A$.heap-size $- 1$
6. \hspace{1em} \textbf{MAX-HEAPIFY} ($A$, 1)
7. \hspace{1em} \textbf{return} $max$

The running time of HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY.

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. An index $i$ into the array identifies the priority-queue element whose key we wish to increase. The procedure first updates the key of element $A[i]$ to its new value. Because increasing the key of $A[i]$ might violate the max-heap property,
the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of 
INSERTION-SORT from Section 2.1, traverses a simple path from this node toward 
the root to find a proper place for the newly increased key. As HEAP-INCREASE-
KEY traverses this path, it repeatedly compares an element to its parent, exchang-
ing their keys and continuing if the element’s key is larger, and terminating if the el-
ement’s key is smaller, since the max-heap property now holds. (See Exercise 6.5-5 
for a precise loop invariant.)

HEAP-INCREASE-KEY \((A, i, key)\)

1. if \(key < A[i]\)
2. error “new key is smaller than current key”
3. \(A[i] = key\)
4. while \(i > 1\) and \(A[\text{PARENT}(i)] < A[i]\)
5. exchange \(A[i]\) with \(A[\text{PARENT}(i)]\)
6. \(i = \text{PARENT}(i)\)

Figure 6.5 shows an example of a HEAP-INCREASE-KEY operation. The running 
time of HEAP-INCREASE-KEY on an \(n\)-element heap is \(O(\lg n)\), since the path 
traced from the node updated in line 3 to the root has length \(O(\lg n)\).

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes 
as an input the key of the new element to be inserted into max-heap \(A\). The proce-
dure first expands the max-heap by adding to the tree a new leaf whose key is \(-\infty\). 
Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct 
value and maintain the max-heap property.

MAX-HEAP-INSERT \((A, key)\)

1. \(A.\text{heap-size} = A.\text{heap-size} + 1\)
2. \(A[A.\text{heap-size}] = -\infty\)
3. HEAP-INCREASE-KEY \((A, A.\text{heap-size}, key)\)

The running time of MAX-HEAP-INSERT on an \(n\)-element heap is \(O(\lg n)\).

In summary, a heap can support any priority-queue operation on a set of size \(n\) 
in \(O(\lg n)\) time.

Exercises

6.5-1
Illustrate the operation of HEAP-EXTRACT-MAX on the heap \(A = (15, 13, 9, 5, 
12, 8, 7, 4, 0, 6, 2, 1)\).
6.5 Priority queues

Figure 6.5  The operation of HEAP-INCREASE-KEY. (a) The max-heap of Figure 6.4(a) with a node whose index is \( i \) heavily shaded. (b) This node has its key increased to 15. (c) After one iteration of the \textbf{while} loop of lines 4–6, the node and its parent have exchanged keys, and the index \( i \) moves up to the parent. (d) The max-heap after one more iteration of the \textbf{while} loop. At this point, \( A[\text{PARENT}(i)] \geq A[i] \). The max-heap property now holds and the procedure terminates.

6.5-2
Illustrate the operation of MAX-HEAP-INSERT \((A, 10)\) on the heap \( A = \{15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\}\).

6.5-3
Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

6.5-4
Why do we bother setting the key of the inserted node to \(-\infty\) in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?
6.5-5
Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the while loop of lines 4–6, the subarray
\( A[1..A.heap-size] \) satisfies the max-heap property, except that there may
be one violation: \( A[i] \) may be larger than \( A[\text{PARENT}(i)] \).

You may assume that the subarray \( A[1..A.heap-size] \) satisfies the max-heap prop-
erty at the time HEAP-INCREASE-KEY is called.

6.5-6
Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires
three assignments. Show how to use the idea of the inner loop of INSERTION-
SORT to reduce the three assignments down to just one assignment.

6.5-7
Show how to implement a first-in, first-out queue with a priority queue. Show
how to implement a stack with a priority queue. (Queues and stacks are defined in
Section 10.1.)

6.5-8
The operation HEAP-DELETE \((A, i)\) deletes the item in node \( i \) from heap \( A \). Give
an implementation of HEAP-DELETE that runs in \( O(\lg n) \) time for an \( n \)-element
max-heap.

6.5-9
Give an \( O(n \lg k) \)-time algorithm to merge \( k \) sorted lists into one sorted list,
where \( n \) is the total number of elements in all the input lists. (Hint: Use a min-
heap for \( k \)-way merging.)

Problems

6-1 Building a heap using insertion
We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the ele-
ments into the heap. Consider the following variation on the BUILD-MAX-HEAP
procedure:
BUILD-MAX-HEAP'(A)
1    A.heap-size = 1
2    for i = 2 to A.length
3        MAX-HEAP-INSERT(A, A[i])

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP' requires \( \Theta(n \lg n) \) time to build an \( n \)-element heap.

6-2 Analysis of d-ary heaps
A \( d \)-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have \( d \) children instead of 2 children.

a. How would you represent a \( d \)-ary heap in an array?

b. What is the height of a \( d \)-ary heap of \( n \) elements in terms of \( n \) and \( d \)?

c. Give an efficient implementation of EXTRACT-MAX in a \( d \)-ary max-heap. Analyze its running time in terms of \( d \) and \( n \).

d. Give an efficient implementation of INSERT in a \( d \)-ary max-heap. Analyze its running time in terms of \( d \) and \( n \).

e. Give an efficient implementation of INCREASE-KEY(\( A, i, k \)), which flags an error if \( k < A[i] \), but otherwise sets \( A[i] = k \) and then updates the \( d \)-ary max-heap structure appropriately. Analyze its running time in terms of \( d \) and \( n \).

6-3 Young tableaus
An \( m \times n \) Young tableau is an \( m \times n \) matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be \( \infty \), which we treat as nonexistent elements. Thus, a Young tableau can be used to hold \( r \leq mn \) finite numbers.

a. Draw a \( 4 \times 4 \) Young tableau containing the elements \{9, 16, 3, 2, 4, 8, 5, 14, 12\}.

b. Argue that an \( m \times n \) Young tableau \( Y \) is empty if \( Y[1,1] = \infty \). Argue that \( Y \) is full (contains \( mn \) elements) if \( Y[m,n] < \infty \).
c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m - 1) \times n$ or an $m \times (n - 1)$ subproblem. (Hint: Think about MAX-HEAPIFY.) Define $T(p)$, where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m + n)$ time bound.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort $n^2$ numbers in $O(n^3)$ time.

f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Divide and Conquer Techniques</td>
<td>Ch 4.3 – 4.6 [1]</td>
</tr>
</tbody>
</table>

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
4.3 Divide and Conquer

The transpose of 2, 4, 1, 5, 3 is 3, 5, 1, 4, 2. Each permutation has a unique transpose and is distinct from its transpose (for \( n > 1 \)). Let \( i \) and \( j \) be integers between 1 and \( n \), and suppose \( j < i \). Then \((i,j)\) is an inversion in exactly one of the permutations \( \pi \) and transpose of \( \pi \). There are \( n(n-1)/2 \) such pairs of integers. Hence each pair of permutations has \( n(n-1)/2 \) inversions between them, and therefore an average of \( n(n-1)/4 \). Thus, overall, the average number of inversions in a permutation is \( n(n-1)/4 \), and we have proved the following theorem.

**Theorem 4.1** Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least \( n(n-1)/2 \) comparisons in the worst case and at least \( n(n-1)/4 \) comparisons on the average (for \( n \) elements).

Since Insertion Sort does \( n(n-1)/2 \) key comparisons in the worst case and approximately \( n^2/4 \) on the average, it is about the best we can do with any algorithm that works "locally," for example, interchanging only adjacent elements. It is, of course, not obvious at this point that any other strategy can do better, but if there are significantly faster algorithms they must move elements more than one position at a time.

4.3 Divide and Conquer

The principle behind the Divide-and-Conquer algorithm design paradigm is that it is (often) easier to solve several small instances of a problem than one large one. Algorithms in Sections 4.4 through 4.8 use the Divide-and-Conquer approach. They divide the problem into smaller instances of the same problem (in this case into smaller sets to be sorted), then solve (conquer) the smaller instances recursively (i.e., by the same method), and finally combine the solutions to obtain the solution for the original input. To escape from the recursion, we solve some small instances of the problem directly. In contrast, Insertion Sort just "chipped off" one element and created one subproblem.

We have already seen one prime example of Divide and Conquer—Binary Search (Section 1.6). The main problem was divided into two subproblems, one of which did not even have to be solved.

In general, we can describe Divide and Conquer by the skeleton procedure in Figure 4.6.

To design a specific Divide-and-Conquer algorithm, we must specify the subroutines directlySolve, divide, and combine. The number of smaller instances into which the input is divided is \( k \). For an input of size \( n \), let \( B(n) \) be the number of steps done by directlySolve, let \( D(n) \) be the number of steps done by divide, and let \( C(n) \) be the number of steps done by combine. Then the general form of the recurrence equation that describes the amount of work done by the algorithm is

\[
T(n) = D(n) + \sum_{i=1}^{k} T(size(i)) + C(n) \quad \text{for } n > \text{smallSize}
\]
Chapter 4  Sorting

\[
\text{solve}(I) \\
\quad n = \text{size}(I); \\
\quad \text{if } (n \leq \text{smallSize}) \\
\quad \quad \text{solution} = \text{directlySolve}(I); \\
\quad \text{else} \\
\quad \quad \text{divide } I \text{ into } I_1, \ldots, I_k, \\
\quad \quad \text{for each } i \in \{1, \ldots, k\}: \\
\quad \quad \quad S_i = \text{solve}(I_i); \\
\quad \quad \text{solution} = \text{combine}(S_1, \ldots, S_k); \\
\quad \text{return solution;}
\]

Figure 4.6 The Divide-and-Conquer skeleton.

with base cases \( T(n) = B(n) \) for \( n \leq \text{smallSize} \). For many Divide-and-Conquer algorithms, either the divide step or the combine step is very simple, and the recurrence equation for \( T \) is simpler than the general form. The Master Theorem (Theorem 3.17) gives solutions for a wide range of Divide-and-Conquer recurrence equations.

QuickSort and MergeSort, the sorting algorithms presented in the next few sections, differ in the ways they divide the problem and later combine the solutions, or sorted subsets. QuickSort is characterized as "hard division, easy combination," while MergeSort is characterized as "easy division, hard combination." Aside from the bookkeeping of procedure calls, we will see that all the "real work" is done in the "hard" section. Both sorting procedures have subroutines to do their "hard" section, and these subroutines are useful in their own rights. For QuickSort, the workhorse is partition, and it is the divide step in the general framework; the combine step does nothing. For MergeSort, the workhorse is merge, and it is the combine step; the divide step does one simple calculation. Both algorithms divide the problem into two subproblems. However, with MergeSort, those two subproblems are of equal size (within a margin of one element), whereas with QuickSort, an even subdivision is not assured. This difference leads to markedly different performance characteristics, which will be discovered during analysis of the respective algorithms.

At the top level, HeapSort (Section 4.8) is not a Divide-and-Conquer algorithm, but uses heap operations that are in the Divide-and-Conquer category. The accelerated form of HeapSort uses a more sophisticated Divide-and-Conquer algorithm.

In later chapters, the Divide-and-Conquer strategy will come up in numerous problems. In Chapter 5, it is applied to the problem of finding the median element of a set. (The general problem is called the selection problem.) In Chapter 6, we will use Divide and Conquer in the form of binary search trees, and their balanced versions, red-black trees. In Chapter 9, we will apply it to problems of paths in graphs, such as transitive closure. In Chapter 12, we will use it on several matrix and vector problems. In Chapter 13, we will apply it to approximate graph coloring. In Chapter 14, it reappears in a slightly different form for parallel computation.

For any query, contact us at help@tutorialsduniya.com
4.4 Quicksort

Quicksort is one of the earlier Divide-and-Conquer algorithms to be discovered; it was published by C. A. R. Hoare in 1962. It is still one of the fastest in practice.

4.4.1 The Quicksort Strategy

Quicksort's strategy is to rearrange the elements to be sorted so that all the "small" keys precede the "large" keys in the array (the "hard division" part). Then Quicksort sorts the two subranges of "small" and "large" keys recursively, with the result that the entire array is sorted. For an array implementation there is nothing to do in the "combination" step, but Quicksort can also work on lists (see Exercise 4.22), in which case the "combination" step concatenates the two lists. We describe the array implementation for simplicity.

Let $E$ be the array of elements and let first and last be the indexes of the first and last entries, respectively, in the subrange Quicksort is currently sorting. At the top level $first = 0$ and $last = n - 1$, where $n$ is the number of elements.

The Quicksort algorithm chooses an element, called the pivot element, whose key is called the pivot, from the subrange that it must sort, and "pulls it out of the way"; that is, it moves the pivot element to a local variable, leaving a vacancy in the array. For the moment we assume that the lefmost element in the subrange is chosen as the pivot element.

Quicksort passes the pivot (the key field only) to the Partition subroutine, which rearranges the other elements, finding an index splitPoint such that:

1. for $first \leq i < splitPoint$, $E[i].key < pivot$;
2. and for $splitPoint < i \leq last$, $E[i].key \geq pivot$.

Notice that there is now a vacancy at splitPoint.

Then Quicksort deposits the pivot element in $E[splitPoint]$, which is its correct position, and the pivot element is ignored in the subsequent sorting. (See Figure 4.7.) This completes the "divide" process, and Quicksort continues by calling itself recursively to solve the two subproblems created by Partition.

The Quicksort procedure may choose to partition around any key in the array between $E[first]$ and $E[last]$, as a preprocessing step. Whatever element is chosen is moved to a local variable named pivot, and if it is not $E[first]$, then $E[first]$ is moved into its position, ensuring that there is a vacancy at $E[first]$ when Partition is called. Other strategies for choosing a pivot are explored in Section 4.4.4.

Algorithm 4.2 Quicksort

Input: Array $E$ and indexes first, and last, such that elements $E[i]$ are defined for $first \leq i \leq last$.

Output: $E[first]$, $\ldots$, $E[last]$ is a sorted rearrangement of the same elements.
4.4.2 The Partition Subroutine

All the work of comparing keys and moving elements is done in the Partition subroutine. There are several different strategies that may be used by Partition; they yield algorithms with different advantages and disadvantages. We present one here and consider another in the exercises. The strategy hinges on how to carry out the rearrangement of elements. A very simple solution is to move elements into a temporary array, but the challenge is to rearrange them in place.

The partitioning method we now describe is essentially the method originally described by Hoare. As motivation, remember that the lower bound argument in Section 4.2.3 showed that, to improve on Insertion Sort, it is necessary to be able to move an element many positions after one compare. Here the vacancy is initially at E[first]. Given that we want small elements at the left end of the range, and that we want to move elements long distances whenever possible, it is very logical to start searching backward from E[last] for a small element, that is, an element less than pivot. When we find one, we move that element into the vacancy (which was at first). That leaves a new vacancy where the small element used to be: we call it highVac. The situation is illustrated in the first two array diagrams in Figure 4.8.
4.4 Quicksort 161

We know all the elements with indexes greater than highVac (through last) are greater than or equal to pivot. If possible, some other large element should be moved into highVac. Again, we want to move elements long distances, so it is logical to search forward for a large element this time, starting at first + 1. When we find one, we move that element into the vacancy (which was at highVac), and that leaves a new vacancy, which we call lowVac. We know all the elements with indexes less than lowVac (down to first) are less than pivot.

Finally, we update the variables low and high as indicated in the last row of Figure 4.8 to prepare for another cycle. As at the beginning of the first cycle, the elements in the range low+1 through high have not been examined yet, and [low] is vacant. We can repeat the cycle just described, searching backward from high for a small element, moving it to the low vacancy, then searching forward from low+1 for a large element, and moving it to highVac, creating a vacancy at lowVac, the position from which the large element was moved. Eventually lowVac and highVac meet, meaning all elements have been compared with the pivot.

The Partition procedure is implemented as a repetition of the cycle just described, using subroutines to organize the code. The subroutine extendLargeRegion scans backward from the right end, passing over large elements until it either finds a small element and moves it into the vacancy at the left end, or runs into that vacancy without finding any small element. In the latter case, the partitioning is completed. In the former case, the new vacant position is returned, and the second subroutine is invoked. The subroutine extendSmallRegion is similar, except that it scans forward from the left end, passing over small elements,
162 Chapter 4 Sorting

until it finds and moves a large element into the vacancy at the right end, or runs out of
data.

Initially, the small-key region (left of low) and large-key region (right of high) are
both empty, and the vacancy is at the left end of the middle region (which is the whole
range at this point). Each call to a subroutine, extendLargeRegion or extendSmallRegion,
shrinks the middle region by at least one, and shifts the vacancy to the other end of the
middle region. The subroutines also ensure that only small elements go into the small-
key region and only large elements go into the large-key region. This can be seen from
their postconditions. When the middle region shrinks to one position, that position is the
vacancy, and it is returned as splitPoint. It is left as an exercise to determine, line by line
in the while loop of partition, what the boundaries are for the middle region and at which
end the vacancy is located. Although the procedure for Partition can "make do" with fewer
variables, each variable we define has its own meaning, and simplifies the answer for the
exercise.

Algorithm 4.3 Partition

Input: Array E, pivot, the key around which to partition, and indexes first, and last, such
that elements E[i] are defined for first + 1 ≤ i ≤ last and E[first] is vacant. It is assumed
that first < last.

Output: Let splitPoint be the returned value. The elements originally in first+1, . . . , last
are rearranged into two subranges, such that

1. the keys of E[first], . . . , E[splitPoint - 1] are less than pivot, and
2. the keys of E[splitPoint+1], . . . , E[last] are greater than or equal to pivot.

Also, first ≤ splitPoint ≤ last, and E[splitPoint] is vacant.

Procedure: See Figure 4.9. ■

To avoid extra comparisons inside the while loop in partition, there is no test for high-
Vac = lowVac before line 5, which would indicate that all elements have been partitioned.
Consequently, high might be one less than low when the loop terminates, when logically it
should be equal. However, high is not accessed after the loop terminates, so this difference
is harmless.

A small example is shown in Figure 4.10. The detailed operation of Partition is shown
only the first time it is called. Notice that the smaller elements accumulate to the left of low
and the larger elements accumulate to the right of high.

4.4.3 Analysis of Quicksort

Worst case

Partition compares each key to pivot, so if there are k positions in the range of the array
it is working on, it does k − 1 key comparisons. (The first position is vacant.) If E[first]
has the smallest key in the range being split then splitPoint = first, and all that has been
accomplished is splitting the range into an empty subrange (keys smaller than pivot) and

For any query, contact us at help@tutorialsduniya.com
4.4 Quicksort  163

```c
int partition(Element[] E, Key pivot, int first, int last)
    int low, high;
1. low = first; high = last;
2. while (low < high)
3.    int highVac = extendLargeRegion(E, pivot, low, high);
4.    int lowVac = extendSmallRegion(E, pivot, low + 1, highVac);
5.    low = lowVac; high = highVac - 1;
6. return low; // This is the splitPoint.

/*/ Postcondition for extendLargeRegion:
* The rightmost element in E[lowVac+1], ..., E[high]
  whose key is < pivot is moved to E[lowVac] and
  the index from which it was moved is returned.
  If there is no such element, lowVac is returned.
*/
int extendLargeRegion(Element[] E, Key pivot, int lowVac, int high)
    int highVac, curr;
    highVac = lowVac; // Assume failure.
    curr = high;
    while (curr > lowVac)
        if (E[curr].key < pivot)
            E[lowVac] = E[curr]; // Succeed.
            highVac = curr;
            break;
        curr --; // Keep looking.
    return highVac;

/*/ Postcondition for extendSmallRegion: (Exercise) */
int extendSmallRegion(Element[] E, Key pivot, int low, int highVac)
    int lowVac, curr;
    lowVac = highVac; // Assume failure.
    curr = low;
    while (curr < highVac)
        if (E[curr].key > pivot)
            E[highVac] = E[curr]; // Succeed.
            lowVac = curr;
            break;
        curr ++; // Keep looking.
    return lowVac;
```

Figure 4.9 Procedure for Algorithm 4.3

For any query, contact us at help@tutorialsduniya.com
Chapter 4  Sorting

The keys
45  14  62  51  75  96  33  84  20

"Pulling out" the pivot
45  14  62  51  75  96  33  84  20

The first execution of Partition

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 62 \quad 51 \quad 75 \quad 96 \quad 33 \quad 84 \quad 20 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{after extendLargeRegion}
\end{align*} \]

beginning of while loop

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 62 \quad 51 \quad 75 \quad 96 \quad 33 \quad 84 \quad 62 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{after extendSmallRegion}
\end{align*} \]

after extendLargeRegion

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 33 \quad 51 \quad 75 \quad 96 \quad 62 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{after extendLargeRegion}
\end{align*} \]

beginning of while loop

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 33 \quad 51 \quad 75 \quad 96 \quad 62 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{after extendSmallRegion}
\end{align*} \]

after extendLargeRegion

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 33 \quad 51 \quad 75 \quad 96 \quad 62 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{after extendSmallRegion}
\end{align*} \]

beginning of while loop

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 33 \quad 51 \quad 75 \quad 96 \quad 62 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{after extendSmallRegion}
\end{align*} \]

after extendSmallRegion

\[ \begin{align*}
\text{low} & : 20 \quad 14 \quad 33 \quad 51 \quad 75 \quad 96 \quad 62 \\
\text{high} & : \text{hVac} \\
\text{hVac} & : \text{white loop exits}
\end{align*} \]

20  14  33  45  75  96  51  84  62  Place pivot in final position

Partition first section (details not shown)
14  20  33

Partition second section (details not shown)
75  14  96  51  84  62

62  51  75  84  96

Partition left subsection
51  62

Partition right subsection
84  96

Final sequence
14  20  33  45  51  62  75  84  96

Figure 4.10  Example of Quicksort

For any query, contact us at help@tutorialsduniya.com
4.4 Quicksort

![Diagram showing the partitioning of an array during the Quicksort algorithm.](image)

**Figure 4.11** Average behavior of Quicksort.

...a subrange with \( k - 1 \) elements. Thus, if pivot is the smallest key each time Partition is called, then the total number of key comparisons done is

\[
\sum_{k=2}^{n} (k - 1) = \frac{n(n - 1)}{2}.
\]

This is as bad as Insertion Sort and Maxsort (Exercise 4.1). And, strangely enough, the worst case occurs when the keys are already sorted in ascending order! Is the name Quicksort just a bit of false advertising?

**Average Behavior**

In Section 4.2.3 we showed that if a sorting algorithm removes at most one inversion from the permutation of the keys after each comparison, then it must do at least \((n^2 - n)/4\) comparisons on the average (Theorem 4.1). Quicksort, however, does not have this restriction. The Partition algorithm can move an element across a large section of the array, eliminating up to \( n - 1 \) inversions with one move. Quicksort deserves its name because of its average behavior.

We assume that the keys are distinct and that all permutations of the keys are equally likely. Let \( k \) be the number of elements in the range of the array being sorted, and let \( A(k) \) be the average number of key comparisons done for ranges of this size. Suppose the next time \( \text{Partition} \) is executed pivot gets put in the \( i \)th position in this subrange (Figure 4.11), counting from 0. Partition does \( k - 1 \) key comparisons, and the subranges to be sorted next have \( i \) elements and \( k - 1 - i \) elements, respectively.

It is important for our analysis that after \( \text{Partition} \) finishes, no two keys within the subrange \((\text{first}, \ldots, \text{splitPoint}-1)\) have been compared to each other, so all permutations of keys in this subrange are still equally likely. The same holds for the subrange \((\text{splitPoint}+1, \ldots, \text{last})\). This justifies the following recurrence.

Each possible position for the split point \( i \) is equally likely (has probability \( 1/k \)) so, letting \( k = n \), we have the recurrence equation

\[
A(n) = n - 1 + \sum_{i=0}^{n-1} \frac{1}{n} (A(i) + A(n - 1 - i)) \quad \text{for } n \geq 2
\]

\[
A(1) = A(0) = 0.
\]
Inspection of the terms in the sum lets us simplify the recurrence equation. The terms of the form \( A(n - 1 - i) \) run from \( A(n - 1) \) down to \( A(0) \), so their sum is the same as the sum of the \( A(i) \) terms. Then we can drop the \( A(0) \) terms, giving

\[
A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \quad \text{for } n \geq 1. 
\] (4.1)

This is a more complicated recurrence equation than the ones we saw earlier, because the value of \( A(n) \) depends on all earlier values. We can try to use some ingenuity to solve the recurrence, or we can make a guess at the solution and prove it by induction. The latter technique is especially suitable for recursive algorithms. It is instructive to see both methods, so we will do both.

To form a guess for \( A(n) \), let's consider a case in which Quicksort works quite well. Suppose that each time Partition is executed, it partitions the range into two equal subranges. Since we're just making an estimate to help guess how fast Quicksort is on the average, we will estimate the size of the two subranges at \( n/2 \) and not worry about whether this is an integer. The number of comparisons done is described by the recurrence equation

\[ Q(n) \approx n + 2Q(n/2). \]

The Master Theorem (Theorem 3.17) can be applied: \( b = 2, c = 2 \), so \( E = 1 \), and \( f(n) = n^1 \). Therefore \( Q(n) \in \Theta(n \log n) \). Thus if \( E \) first were close to the median each time the range is split, the number of comparisons done by Quicksort would be in \( \Theta(n \log n) \). This is significantly better than \( \Theta(n^2) \). But if all permutations of the keys are equally likely, are there enough "good" cases to affect the average? We prove that there are.

**Theorem 4.2** Let \( A(n) \) be defined by the recurrence equation Equation (4.1). Then, for \( n \geq 1 \), \( A(n) \leq cn \ln n \) for some constant \( c \). (Note: We have switched to the natural logarithm to simplify some of the computation in the proof. The value for \( c \) will be found in the proof.)

**Proof** The proof is by induction on \( n \), the number of elements to be sorted. The base case is \( n = 1 \). We have \( A(1) = 0 \) and \( c1 \ln 1 = 0 \).

For \( n > 1 \), assume that \( A(i) \leq ci \ln(i) \) for \( 1 \leq i < n \), for the same constant \( c \) stated in the theorem. By Equation (4.1) and the induction hypothesis,

\[
A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i) \leq n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} ci \ln(i).
\]

We can bound the sum by integrating (see Equation 1.16):

\[
\sum_{i=1}^{n-1} ci \ln(i) \leq c \int_{1}^{n} x \ln x \, dx.
\]

Using Equation (1.15) from Section 1.3.2 gives
4.4 Quicksort 167

\[ \int_1^n x \ln x \, dx = \frac{1}{2} n^2 \ln(n) - \frac{1}{4} n^2 \]

so

\[ A(n) = n - 1 + \frac{2c}{n} \left( \frac{1}{2} n^2 \ln(n) - \frac{1}{2} n^2 \right) \]

\[ = cn \ln n + n(1 - \frac{c}{2}) - 1. \]

To show that \( A(n) \leq cn \ln n \), it suffices to show that the second and third terms are negative or zero. The second term is less than or equal to zero for \( c \geq 2 \). So we can let \( c = 2 \) and conclude that \( A(n) \leq 2n \ln n \). \( \Box \)

A similar analysis shows that \( A(n) > cn \ln n \) for any \( c < 2 \). Since \( \ln n \approx 0.693 \lg n \), we therefore have:

**Corollary 4.3** On average, assuming all input permutations are equally likely, the number of comparisons done by Quicksort (Algorithm 4.2) on sets of size \( n \) is approximately \( 1.386 n \lg n \), for large \( n \). \( \Box \)

**Average Behavior, More Exactly**

Although we have established the average behavior of Quicksort, it is still instructive to return to the recurrence equation (Equation 4.1) and try to solve it directly, getting more than the leading term. This section uses some sophisticated mathematics, and can be omitted without loss of continuity.

We have, by Equation (4.1),

\[ A(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} A(i). \]  \hspace{1cm} (4.2)

\[ A(n-1) = n - 2 + \frac{2}{n-1} \sum_{i=1}^{n-2} A(i). \]  \hspace{1cm} (4.3)

If we subtract the summation in Equation (4.3) from the summation in Equation (4.2), most of the terms drop out. Since the summations are multiplied by different factors, we need a slightly more complicated bit of algebra. Informally, we compute

\[ n \times \text{Equation (4.2)} - (n-1) \times \text{Equation (4.3)}. \]

So

\[ nA(n) - (n-1)A(n-1) = n(n-1) + 2 \sum_{i=1}^{n-1} A(i) - (n-1)(n-2) - 2 \sum_{i=1}^{n-2} A(i) \]

\[ = 2A(n-1) + 2(n-1). \]

For any query, contact us at help@tutorialsduniya.com
Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.

168 Chapter 4 Sorting

So

\[ A(n) = \frac{A(n-1)}{n+1} + \frac{2(n-1)}{n(n+1)} \]

Now let

\[ B(n) = \frac{A(n)}{n+1} \]

The recurrence equation for \( B \) is

\[ B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)} \quad B(1) = 0. \]

With the aid of Equation (1.11), we leave it for readers to verify that

\[ B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2 \sum_{i=1}^{n} \frac{1}{i} - 4 \sum_{i=1}^{n} \frac{1}{i(i+1)} \]

\[ \approx 2(\ln n + 0.577) - 4n/(n+1). \]

Therefore

\[ A(n) \approx 1.386 n \log n - 2.846 n. \]

Space usage

At first glance it may seem that Quicksort is an in-place sort. It is not. While the algorithm is working on one subrange, the beginning and ending indexes (call them the borders) of all the other subranges yet to be sorted are saved on the frame stack, and the size of the stack depends on the number of subranges into which the range will be split. This, of course, depends on \( n \). In the worst case, Partition splits off one entry at a time; the depth of the recursion is \( n \). Thus the worst-case amount of space used by the stack is in \( \Theta(n) \). One of the modifications to the algorithm described next can significantly reduce the maximum stack size.

4.4.4 Improvements on the Basic Quicksort Algorithm

Choice of Pivot

We have seen that Quicksort works well if the pivot key used by Partition to partition a segment belongs near the middle of the segment. (Its position is the value, splitPoint, returned by Partition.) Choosing \( E[\text{first}] \) as the pivot element causes Quicksort to do poorly in cases where sorting should be easy (for example, when the array is already sorted). There are several other strategies for choosing the pivot element. One is to choose a random integer \( q \) between first and last and let pivot = \( E[q] \). Another is to let pivot be the median key of the entries \( E[\text{first}], E[\text{first}+\text{last}/2], \) and \( E[\text{last}] \). (In either case, the element in \( E[\text{first}] \) would be swapped with the pivot element before proceeding with the Partition.

For any query, contact us at help@tutorialsduniya.com
algorithm.) Both of these strategies require some extra work to choose pivot, but they pay off by improving the average running time of a Quicksort program.

Alternative Partition Strategy

The version of Partition presented in the text does the fewest element movements, on average, compared to other partitioning strategies. It is shown with subroutines for clarity, and coding these in-line would save some overhead; however, some optimizing compilers can make this change automatically. Other optimization considerations are mentioned in Notes and References at the end of the chapter. There is an alternative version in the exercises that is easy to understand and program, but somewhat slower.

Small Sort

Quicksort is not particularly good for small sets, due to the overhead of procedure calls. But, by the nature of the algorithm, for large \( n \) Quicksort will break the set up into small subsets and recursively sort them. Thus whenever the size of a subset is small, the algorithm becomes inefficient. This problem can be remedied by choosing a small smallSortSize and sorting subsets of size less than or equal to smallSortSize by some simple, nonrecursive sort, called smallSort in the modified algorithm. (Insertion Sort is a good choice.)

```c
quickSort(E, first, last)
    if (last - first > smallSortSize)
        pivotElement = E[first];
        pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first, last);
        E[splitPoint] = pivotElement;
        quickSort(E, first, splitPoint - 1);
        quickSort(E, splitPoint + 1, last);
    else
        smallSort(E, first, last);
```

A variation on this theme is to skip calling smallSort. Then when Quicksort exits, the array is not sorted, but no element needs to move more than smallSortSize places to reach its correct sorted position. (Why not?) Therefore one postprocessing run of Insertion Sort will be very efficient, and will do about the same comparisons as all the calls to it in its role as smallSort.

What value should smallSortSize have? The best choice depends on the particular implementation of the algorithm (that is, the computer being used and the details of the program), since we are making some trade-offs between overhead and key comparisons. A value close to 10 may do reasonably well.

Stack Space Optimization

We observed that the depth of recursion for Quicksort can grow quite large, proportional to \( n \) in the worst case (when Partition splits off only one element each time). Much of the pushing and popping of the frame stack that will be done is unnecessary. After Partition,
Chapter 4  Sorting

the program starts sorting the subrange \(E[\text{first}], \ldots, E[\text{splitPoint} - 1]\); later it must sort the subrange \(E[\text{splitPoint} + 1], \ldots, E[\text{last}]\).

The second recursive call is the last statement in the procedure, so it can be converted into iteration in the manner we have seen earlier for shiftVac in Insertion Sort. The first recursive call remains, so the recursion is only partially eliminated.

With only one recursive call left in the procedure, we still need to be concerned about excessive depth of recursion. This can occur through a succession of recursive calls that each work on a subrange only slightly smaller than the preceding one. So the second trick we use is to avoid making the recursive call on the larger subrange. By ensuring that each recursive call is on at most half as many elements as its “parent” call, the depth of recursion is guaranteed to remain within about \(\log n\). The two ideas are combined in the following version, in which “TRO” stands for “tail recursion optimization.” The idea is that after each partition, the next recursive call will work on the smaller subrange, and the larger subrange will be handled directly in the while loop.

```c
quickSortTRO(E, first, last)

    int first1, last1, first2, last2;
    first2 = first; last2 = last;
    while (last2 - first2 > 1)
        pivotElement = E[first];
        pivot = pivotElement.key;
        last splitPoint = partition(E, pivot, first2, last2);
        E[splitPoint] = pivotElement;
        if (splitPoint < (first2 + last2) / 2)
            first1 = first2; last1 = splitPoint - 1;
            first2 = splitPoint + 1; last2 = last2;
        else
            first1 = splitPoint + 1; last1 = last2;
            first2 = first2; last2 = splitPoint - 1;
        quickSortTRO(E, first1, last1);
    // Continue loop for first2, last2.
    return;
```

Combined Improvements
We discussed the preceding modifications independently, but they are compatible and can be combined in one program.

Remarks
In practice, Quick sort programs run quite fast on the average for large \(n\), and they are widely used. In the worst case, though, Quick sort behaves poorly. Like Insertion Sort (Section 4.2), Maxisort and Bubble Sort (Exercises 4.1 and 4.2), Quick sort’s worst-case time is in \(\Theta(n^2)\), but unlike the others, Quick sort’s average behavior is in \(\Theta(n \log n)\). Are there
4.5 Merging Sorted Sequences

In this section we review a straightforward solution to the following problem: Given two sequences \( A \) and \( B \) sorted in nondecreasing order, merge them to create one sorted sequence \( C \). Merging sorted subsequences is essential to the strategy of Mergesort. It also has numerous applications in its own right, some of which are covered in the exercises. The measure of work done by a merge algorithm will be the number of comparisons of keys performed by the algorithm.

Let \( k \) and \( m \) be the number of items in sequences \( A \) and \( B \), respectively. Let \( n = k + m \) be the "problem size." Assuming neither \( A \) nor \( B \) is empty, we can immediately determine the first item in \( C \): It is the minimum between the first items of \( A \) and \( B \). What about the rest of \( C \)? Suppose the first element of \( A \) was the minimum. Then the remainder of \( C \) must be the result of merging all elements of \( A \) after the first with all elements of \( B \). But this is just a smaller version of the same problem we started with. The situation is symmetrical if the first element of \( B \) was the minimum. In either case the problem size for the remaining problem (of constructing the rest of \( C \)) is \( n - 1 \). Method 99 (Section 3.2.2) comes to mind.

If we assume that we only need to merge problems of size up to 100, and we can call upon merge99 to merge problems of size up to 99, then the problem is already solved. The pseudocode follows:

```plaintext
merge(A, B, C)
if (A is empty)
    rest of C = rest of B
else if (B is empty)
    rest of C = rest of A
else
    if (first of A is less than first of B)
        first of C = first of A
        merge99(rest of A, B, rest of C)
    else
        first of C = first of B
        merge99(A, rest of B, rest of C)
return
```

Now just change merge99 to merge for the general recursive solution.

Once the solution idea is seen, we can also see how to formulate an iterative solution. The idea works for all sequential data structures, but we state the algorithm in terms of
Chapter 4  Sorting

arrays, for definiteness. We introduce three indexes to keep track of where "rest of $A$," "rest of $B$," and "rest of $C$" begin at any stage in the iteration. (These indexes would be parameters in the recursive version.)

Algorithm 4.4  Merge

Input: Arrays $A$ with $k$ elements and $B$ with $m$ elements, each in nondecreasing order of their keys.

Output: $C$, an array containing $n = k + m$ elements from $A$ and $B$ in nondecreasing order. $C$ is passed in and the algorithm fills it.

void merge(Element[] A, int k, Element[] B, int m, Element[] C)

int indexA = 0; indexB = 0; indexC = 0;
// indexA is the beginning of rest of A, same for B, C.

while (indexA < k && indexB < m)
if (A[indexA].key < B[indexB].key)
    C[indexC] = A[indexA];
    indexA ++;
    indexC ++;
else
    C[indexC] = B[indexB];
    indexB ++;
    indexC ++;
// Continue loop
if (indexA >= k)
    Copy B[indexB, ..., m-1] to C[indexC, ..., n-1].
else
    Copy A[indexA, ..., k-1] to C[indexC, ..., n-1].

4.5.1  Worst Case

Whenever a comparison of keys from $A$ and $B$ is done, at least one element is moved to $C$ and never examined again. After the last comparison, at least two elements—the two just compared—have not yet been moved to $C$. The smaller one is moved immediately, but now $C$ has at most $n - 1$ elements, and no more comparisons will be done. Those that remain in the other array are moved to $C$ without any further comparisons. So at most $n - 1$ comparisons are done. The worst case, using all $n - 1$ comparisons, occurs when $A[k-1]$ and $B[m-1]$ belong in the last two positions in $C$.

4.5.2  Optimality of Merge

We show next that Algorithm 4.4 is optimal in the worst case among comparison-based algorithms when $k = m = n/2$. That is, for any comparison-based algorithm that merges
correctly on all inputs for which \( k = m = n/2 \) there must be some input for which it requires \( n - 1 \) comparisons. (This is not to say that for a particular input no algorithm could do better than Algorithm 4.4.) After considering \( k = m = n/2 \), we look at some other relationships between \( k \) and \( m \).

Theorem 4.4 Any algorithm to merge two sorted arrays, each containing \( k = m = n/2 \) entries, by comparison of keys, does at least \( n - 1 \) such comparisons in the worst case.

Proof Suppose we are given an arbitrary merge algorithm. Let \( a_i \) and \( b_i \) be the \( i \)th entries of \( A \) and \( B \), respectively. We show that keys can be chosen so that the algorithm must compare \( a_i \) with \( b_i \) for \( 0 \leq i < m \), and \( a_i \) with \( b_{i+1} \), for \( 0 \leq i < m - 1 \). Specifically, choose keys so that, whenever the algorithm compares \( a_i \) and \( b_j \), if \( i < j \), the result is that \( a_i < b_j \), and if \( i \geq j \), the result is that \( b_j < a_i \). Choosing the keys so that

\[
b_0 < a_0 < b_1 < a_1 < \cdots < b_{i+1} < a_{i+1} < \cdots < b_{m-1} < a_{m-1}
\]  

(4.4)

will satisfy these conditions. However, if for some \( i \), the algorithm never compares \( a_i \) and \( b_i \), then choosing keys in the same order as in Equation (4.4), except that \( a_i < b_i \), will also satisfy these conditions and the algorithm would not be able to determine the correct ordering.

Similarly, if for some \( i \), it never compares \( a_i \) and \( b_{i+1} \), the arrangement of Equation (4.4), except that \( b_{i+1} < a_i \), would be consistent with the results of the comparisons done, and again the algorithm could not determine the correct ordering. \( \square \)

Can we generalize this conclusion? Suppose \( k \) and \( m \) differ slightly (as we will see they might in Mergesort)?

Corollary 4.5 Any algorithm to merge two sorted arrays by comparison of keys, where the inputs contain \( k \) and \( m \) entries, respectively, \( k \) and \( m \) differ by one, and \( n = k + m \), does at least \( n - 1 \) such comparisons in the worst case.

Proof The same proof as in Theorem 4.4 applies, except there is no \( a_{m-1} \). \( \square \)

Can we generalize this conclusion still further? If we find one kind of behavior at one extreme, it is often a good idea to check the other extreme. Here the first "extreme" was \( k = m \), so the other extreme makes \( k \) and \( m \) as different as possible. Let's look at an extreme case, where \( k = 1 \) and \( m \) is large, so \( n = m + 1 \). We can devise an algorithm that uses at most \( \lceil \log(n+1) \rceil \) comparisons. (What is it?) So clearly \( n - 1 \) is not a lower bound in this case. The improvement for \( k = 1 \) can be generalized to other cases where \( k \) is much less than \( n \) (see Exercise 4.24). Therefore the lower bound arguments of Theorem 4.4 and Corollary 4.5 cannot be extended to all combinations of \( k \) and \( m \). For further possibilities, look at Exercise 4.33 after reading Section 4.7.

4.5.3 Space Usage

It might appear from the way in which Algorithm 4.4 is written that merging sequences with a total of \( n \) entries requires enough memory locations for \( 2n \) entries, since all entries...
are copied to C. In some cases, however, the amount of extra space needed can be decreased. One case is that the sequences are linked lists, and A and B are not needed (as lists) after the merge is completed. Then the list nodes of A and B can be recycled as C is created.

Suppose the input sequences are stored in arrays and suppose \( k \geq m \). If A has enough room for \( n = k + m \) elements, then only the extra \( m \) locations in A are needed. Simply identify C with A, and do the merging from the right ends (larger keys) of A and B, as indicated in Figure 4.12. The first \( m \) entries moved to "C" will fill the extra locations of A. From then on the vacated locations in A are used. There will always be a gap (i.e., some empty locations) between the end of the merged portion of the array and the remaining entries of A until all of the entries have been merged. Observe that if this space-saving storage layout is used, the last lines in the merge algorithm (else CopyA[indexA], \( \ldots \), A[k-1] to C[indexC], \( \ldots \), C[n-1]) can be eliminated because, if B empties before A, the remaining items in A are in their correct position and do not have to be moved.

Whether or not C overlaps one of the input arrays, the extra space used by the Merge algorithm when \( k = m = n/2 \) is in \( \Theta(n) \).

4.6 Mergesort

The problem with Quicksort is that Partition doesn't always decompose the array into two equal subranges. Mergesort just slices the array in two halves and sorts the halves separately (and of course, recursively). Then it merges the sorted halves (see Figure 4.13).
Thus, using the divide-and-conquer terminology of Section 4.3, divide merely computes the middle index of the subrange and does no key comparisons; combine does the merging.

We assume that Merge is modified to merge adjacent subranges of one array, putting the resulting merged array back into the cells originally occupied by the elements being merged. Its parameters now are the array name E, and the first, mid, and last indexes of the subranges it is to merge; that is, the sorted subranges are E[first], . . . , E[mid] and E[mid+1], . . . , E[last], and the final sorted range is to be E[first], . . . , E[last]. In this modification, the merge subroutine is also responsible for allocating additional workspace needed. Some of the issues were discussed in Section 4.5.3.

Algorithm 4.5 Mergesort

Input: Array E and indexes first, and last, such that the elements of E[i] are defined for first ≤ i ≤ last.

Output: E[first], . . . , E[last] is a sorted rearrangement of the same elements.

void mergeSort(Element[] E, int first, int last)
if (first < last)
    Int mid = (first+last) / 2;
    mergeSort(E, first, mid);
    mergeSort(E, mid+1, last);
    merge(E, first, mid, last);
    return;
We have observed that students often confuse the Merge and Mergesort algorithms. Remember that Mergesort is a sorting algorithm. It begins with one scrambled array and sorts it. Merge begins with two arrays that are already sorted, it combines them into one sorted array.

**Mergesort Analysis**

First, we find the asymptotic order of the worst-case number of key comparisons for Mergesort. As usual, we define the problem size as \( n = \text{fast} - \text{first} + 1 \), the number of elements in the range to be sorted. The recurrence equation for the worst-case behavior of Mergesort is

\[
W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \tag{4.5}
\]

\[ W(1) = 0. \]

The Master Theorem tells us immediately that \( W(n) \in \Theta(n \log n) \). So we finally have a sorting algorithm whose worst-case behavior is in \( \Theta(n \log n) \). Rather than carry out a separate analysis of the average complexity of Mergesort, we will defer this question until we have developed the very general Theorem 4.11, concerning average behavior, in Section 4.7, just ahead.

A possible disadvantage of Mergesort is its requirement for auxiliary workspace. Because of the extra space used for the merging, which is in \( \Theta(n) \), Mergesort is not an in-place sort.

* **Mergesort Analysis, More Exactly**

It is of some interest to obtain a more exact estimate of the worst-case number of comparisons, in light of lower bounds to be developed in the next section (Section 4.7). We will see that Mergesort is very close to the lower bound. Readers may skip the details of this section without loss of continuity and proceed to its main conclusion, Theorem 4.6.

In the recursion tree for Equation (4.5) (see Figure 4.14), we observe that the nonrecursive costs of nodes at depth \( d \) sum to \( n - 2^d \) (for all node depths not containing any base cases). We can determine that all base cases (for which \( W(1) = 0 \)) occur at depths \( \lfloor \log(n+1) \rfloor - 1 \) or \( \lceil \log(n+1) \rceil \). There are exactly \( n \) base-case nodes. Let the maximum depth be \( D \) (that is, \( D = \lceil \log(n+1) \rceil \)) and let \( B \) be the number of base cases at depth \( D - 1 \). Then there are \( n - B \) base cases at depth \( D \) (and no other nodes at depth \( D \)). Each nonbase node at depth \( D - 1 \) has two children, so there are \( (n - B)/2 \) nonbase cases at depth \( D - 1 \). Using this information, we compute the sum of nonrecursive costs for the last few depths as follows:

1. Depth \( D - 2 \) has \( 2^{D-2} \) nodes, none of which are base cases. The sum of nonrecursive costs for this level is \( n - 2^{D-2} \).
2. Depth \( D - 1 \) has \( (n - B)/2 \) nonbase cases. Each has problem size 2 (with cost 1), so the sum of the nonrecursive costs for this level is \( (n - B)/2 \).
3. Depth \( D \) has \( n - B \) base cases, cost 0.

For any query, contact us at help@tutorialsduniya.com
You can verify that $B = 2^D - n$ (Exercise 4.29). Therefore

$$W(n) = \sum_{d=0}^{D-2} (n - 2^d) + (n - B)/2$$

$$= n(D - 1) - 2^{D-1} + 1 + (n - B)/2$$

$$= n D - 2^D + 1,$$  (4.6)

Because $D$ is rounded up to an integer and occurs in the exponent, it is hard to tell how Equation (4.6) behaves between powers of 2. We prove the following theorem, which removes the ceiling function from the exponent.

**Theorem 4.6** The number of comparisons done by Mergesort in the worst case is between $[n \log(n) - n + 1]$ and $[n \log(n) - .914 n]$.

**Proof** If we define $\alpha = 2^D/n$, then $1 \leq \alpha < 2$, and $D$ can be replaced throughout Equation (4.6) by $(\log(n) + \log(\alpha))$. This leads to $W(n) = n \log(n) - (\alpha - \log(\alpha))n + 1$. The minimum value of $(\alpha - \log(\alpha))$ is about .914 (see Exercise 4.30) and the maximum in the range under consideration is 1.

Thus Mergesort does about 30 percent fewer comparisons in the worst case than Quicksort does in the average case. However, Mergesort does more element movement than Quicksort does on average, so its time may not be faster (see Exercises 4.21 and 4.27).
List combine(List[] buckets, int radix)
   
   // Combine linked lists in all buckets into one list L.
   int b; // bucket number
   List L, remBucket;

   L = nil;
   for (b = radix - 1; b >= 0; b --)
      remBucket = buckets[b];
   while (remBucket != nil)
      Key K = first(remBucket);
      L = cons(K, L);
      remBucket = rest(remBucket);
   return L;

Analysis and Remarks

Distributing one key requires extracting a field and doing a few link operations; the number of steps is bounded by a constant. So, for all keys, distribute does \( \Theta(n) \) steps. Similarly, combine does \( \Theta(n) \) steps. The number of distribution and combination passes is number of fields, the number of fields used for distribution. If this can be held constant, the total number of steps done by Radix Sort is linear in \( n \).

Our implementation of Radix Sort used \( \Theta(n) \) extra space for the link fields, provided the radix is bounded by \( n \). Other implementations that do not use links also use extra space in \( \Theta(n) \).

Exercises

Section 4.1 Introduction

4.1 One of the easiest sorting algorithms to understand is one that we call Maxsort. It works as follows: find the largest key, say \( max \), in the unsorted section of the array (initially the whole array) and then interchange \( max \) with the element in the last position in the unsorted section. Now \( max \) is considered part of the sorted section consisting of larger keys at the end of the array; it is no longer in the unsorted section. Repeat this procedure until the whole array is sorted.

a. Write an algorithm for Maxsort assuming an array \( E \) contains \( n \) elements to be sorted, with indexes 0, . . . , \( n - 1 \).

b. How many comparisons of keys does Maxsort do in the worst case? On the average?

4.2 The next few exercises are about a sorting method called Bubble Sort. It sorts by making several passes through the array, comparing pairs of keys in adjacent locations, and interchanging their elements if they are out of order. That is, the first and second keys are compared and interchanged if the first is larger than the second; then the (new) second
and the third keys are compared and interchanged if necessary, and so on. It is easy to see that the largest key will bubble up to the end of the array; on subsequent passes it will be ignored. If on any pass no entries are interchanged, the array is completely sorted and the algorithm can halt. The following algorithm makes this informal description of the method precise.

Algorithm 4.14 Bubble Sort

Input: E, an array of elements, and \( n \geq 0 \), the number of elements.

Output: E with elements in nondecreasing order of their keys.

```java
void bubbleSort(Element[] E, int n)
    int numPairs; // the number of pairs to be compared
    boolean didSwitch; // true if an interchange is done
    int j;

    numPairs = n - 1;
    didSwitch = true;
    while (didSwitch)
        didSwitch = false;
        for (j = 0; j < numPairs; j++)
            if (E[j] > E[j + 1])
                Interchange E[j] and E[j + 1];
                didSwitch = true;
                // Continue for loop.
        numPairs -=;
    return;
```

The example in Figure 4.29 illustrates how Bubble Sort works.

![Figure 4.29 Bubble Sort](image)

For any query, contact us at help@tutorialsduniya.com
a. How many key comparisons does Bubble Sort do in the worst case? What arrangement of keys is a worst case?
b. What arrangement of keys is a best case for Bubble Sort, that is, for what input does it do the fewest comparisons? How many comparisons does it do in the best case?

4.3 The correctness of Bubble Sort (Exercise 4.2) depends on several facts. These are easy to verify but worth doing in order to consciously recognize the mathematical properties involved.
a. Prove that, after one pass through the array, the largest entry will be at the end.
b. Prove that, if there is no pair of consecutive entries out of order, then the entire array is sorted.

4.4 We can modify Bubble Sort (Exercise 4.2) to avoid unnecessary comparisons in the tail of the array by keeping track of where the last interchange occurred in the for loop.

a. Prove that if the last exchange made in some pass occurs at the \( j \)th and \((j + 1)\)st positions, then all entries from the \((j + 1)\)st through the \((n - 1)\)th are in their correct position. (Note that this is stronger than saying simply that these items are in order.)
b. Modify the algorithm so that if the last exchange made in a pass occurs at the \( j \)th and \((j + 1)\)st positions, the next pass will not examine any entries from the \((j + 1)\)st position to the end of the array.
c. Does this change affect the worst-case behavior of the algorithm? If so, how?

4.5 Can something similar to the improvement in the preceding exercise be done to avoid unnecessary comparisons when the keys at the beginning of the array are already in order? If so, write the modifications to the algorithm. If not, explain why not.

Section 4.2 Insertion Sort

4.6 We observed that a worst case for Insertion Sort occurs when the keys are initially in decreasing order. Describe at least two other initial arrangements of the keys that are also worst cases. Show inputs for which the exact number of key comparisons (not just the asymptotic order) is the worst possible.

4.7 What is a best case for Insertion Sort? Describe how the elements in the list would be arranged, and tell how many comparisons of list elements would be done in that case.

4.8 Consider the following variation of Insertion Sort: For \( 1 \leq i < n \), to insert the element \( E[i] \) among \( E[0] \leq E[1] \leq \cdots \leq E[i - 1] \), do a Binary Search to find the correct position for \( E[i] \).

a. How many key comparisons would be done in the worst case?
b. How many times are elements moved in the worst case?
c. What is the asymptotic order of the worst-case running time?
d. Can the number of moves be reduced by putting the elements in a linked list instead of an array? Explain.

4.9 In the average analysis of Insertion Sort we assumed that the keys were distinct. Would the average for all possible inputs, including cases with duplicate keys, be higher or lower? Why?

4.10 Show that a permutation on \( n \) items has at most \( n(n-1)/2 \) inversions. Which permutation(s) have exactly \( n(n-1)/2 \) inversions?

4.11 Give an algorithm to perform Insertion Sort on a linked list of integers, using the operations of the IntList abstract data type, in Section 2.3.2. Analyze its time and space requirements. Does the space usage depend on whether the language has “garbage collection” (see Example 2.1)?

Section 4.3 Divide and Conquer

4.12 Suppose we have a straightforward algorithm for a problem that does \( \Theta(n^2) \) steps for inputs of size \( n \). Suppose we devise a Divide-and-Conquer algorithm that divides an input into two inputs half as big, and does \( O(n \log n) \) steps to divide the problem and \( C(n) = n \log n \) steps to combine the solutions to get a solution for the original input. Is the Divide-and-Conquer algorithm more or less efficient than the straightforward scheme? Justify your answer. Hint: See Equation (3.14) and Exercise 3.10.

Section 4.4 Quicksort

4.13 Complete the postconditions of extendSmallRegion in Algorithm 4.3.

4.14 In Algorithm 4.3, define the middle region to be the range of indexes containing the unexamined elements and the vacancy. For each of the lines 2 through 5 in the partition procedure, which variables or variable expressions (some plus or minus 1's might be needed) specify the left and right ends of the middle region? For each of the lines 2 through 5, which end of the middle region contains the vacancy? Answer for the situation immediately before each line is executed.

4.15 How many key comparisons does Quicksort (Algorithms 4.2 and 4.3) do if the array is already sorted? How many element movements does it do?

4.16 Prove that if the “stack space optimization” improvement in Section 4.4.4 is used in Algorithm 4.2, then the maximum stack size is in \( O(\log n) \).

4.17 Suppose that, instead of choosing \( E[\text{first}] \) as pivot, Quicksort lets pivot be the median of \( E[\text{first}], E[\text{first}+\text{last}/2], \) and \( E[\text{last}] \). How many key comparisons will Quicksort do in the worst case to sort \( n \) elements? (Remember to count the comparisons done in choosing pivot.)
Chapter 4  Sorting

4.18 This exercise examines an alternative algorithm for Partition, with simple and elegant code. This method is due to Lamport; we call it partitionL. The idea, as illustrated in Figure 4.30, is to collect small elements to the left of the vacancy, large elements immediately to the right of the vacancy, with unknown (i.e., unexamined) elements being at the far right of the range. Initially, all the elements are in the unknown group. "Small" and "large" are determined with respect to pivot. When partitionL finds a small element in the unknown group, it moves the small element to the vacancy, then creates a new vacancy one place to the right by moving a large element from that place to the end of the "large" range.

```c
int partitionL(Element[] E, Key pivot, int first, int last)
    int vacant, unknown;
    vacant = first;
    for (unknown = first+1; unknown <= last; unknown ++)
        if (E[unknown] < pivot)
            E[vacant] = E[unknown];
        E[unknown] = E[vacant+1];
        vacant ++;
    return vacant;
```

At each iteration of its loop partitionL compares the next unknown element, which is E[unknown], to pivot. Finally, after all the elements have been compared to pivot, vacant is returned as the splitPoint.

a. At the beginning of each of the lines 2 through 6, what are the boundaries of the small-key region and the large-key region? Express your answer using unknown and other index variables.

b. At the beginning of line 7, what are the boundaries of the small-key region and the large-key region? Express your answer without using unknown.

c. How many key comparisons does partitionL do on a subrange of E with k elements? If QuickSort uses partitionL instead of partition, what is the impact on the total number of key comparisons done in the worst case?

4.19 Suppose the array E contains the keys 10, 9, 8, ..., 2, 1, and is to be sorted using QuickSort.

a. Show how the keys would be arranged after each of the first two calls to the partition procedure in Algorithm 4.3. Tell how many element movements are done by each of these two calls to partition. From this example, estimate the total number of element movements that would be done to sort n elements initially in decreasing order.

b. Do the same for partitionL described in the preceding exercise.

c. List some of the relative advantages and disadvantages of the two partition algorithms.
4.20 Suppose all \( n \) elements in the array to be sorted by Quick sort are equal. How many key comparisons will Quick sort do to sort the array (using partition in Algorithm 4.3)? Justify your answer.

4.21 This exercise explores the average number of element movements done by Quick sort using different versions of Partition. Hint for parts (a) and (b): When an element is compared with the pivot, what is the probability that it has to be moved?

a. How many element movements does Quick sort do on the average, using the partition1 subroutine of Exercise 4.18?

b. How many element movements does Quick sort do on the average, using the partition subroutine of Algorithm 4.3?

c. How do these results compare to Mergesort (see Exercise 4.27)?

4.22 Write a version of Quick sort and Partition for linked lists of integers, using the operations of the IntList abstract data type, in Section 2.3.2. Analyze its time and space
Chapter 4  Sorting

requirements. Does the space usage depend on whether the language has "garbage collection" (see Example 2.1)?

Section 4.5 Merging Sorted Sequences

4.23 Give an algorithm to merge two sorted linked lists of integers, using the operations of the IntList abstract data type, in Section 2.3.2.

4.24 Suppose the array subranges to be merged are of lengths $k$ and $m$, where $k$ is much less than $m$. Describe a merging algorithm that takes advantage of this to do at most (say) $(k + m)/2$ comparisons, provided $k$ is sufficiently small in relation to $m$. How small does $k$ have to be to achieve this bound in the worst case? Is there a range of $k$ for which the bound $\sqrt{k + m}$ can be achieved? What can you say about the amount of element movement that is needed in these cases?

4.25 Show that the number of permutations that can be formed by merging two sorted segments $A$ and $B$ of lengths $k$ and $m$, where $k + m = n$, is \( \binom{n}{2} \). (Recall this notation from Equation 1.1.) Assume $k \leq m$ for definiteness, and assume no duplicate keys. Hint: Formulate a recurrence based on the relationship of $A[0]$ and $B[0]$, then look at Exercise 1.2. There are several other ways of looking at this problem that also work.

Section 4.6 Mergesort

4.26 How many key comparisons are done by Mergesort if the keys are already in order when the sort begins?

4.27 Mergesort (Algorithm 4.5) was described assuming that Merge developed its output in a work array, then copied the contents of the work array back to the input array when it was done.

a. Work out a strategy for "toggling" between the input array and the work array to avoid this extra copying. That is, at alternate levels of the recursion either the original input array has the data to be merged or the work array has it.

b. With the above optimization, how many element movements does Mergesort do on the average? How does this compare to Quicksort (see Exercise 4.21)?

4.28 Write a version of Mergesort for linked lists of integers, using the operations of the IntList abstract data type, in Section 2.3.2. Analyze its time and space requirements. Does the space usage depend on whether the language has "garbage collection" (see Example 2.1)?

4.29 For the Mergesort analysis using the recursion tree (Section 4.6), where $D$ is the maximum depth of the tree and $B$ is the number of base cases at depth $D - 1$, verify that $B = 2^D - n$.

4.30 Derive the minimum value of the expression $(\alpha - \lg \alpha)$ in the interval $(1,2)$, which was used in the proof of Theorem 4.6. Show that it is $(1 + \ln \ln 2)/\ln 2$.

For any query, contact us at help@tutorialsduniya.com
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>Linear time Sorting Algorithms</td>
<td>Ch 8 [2]</td>
</tr>
<tr>
<td></td>
<td>Medians and Order Statistics</td>
<td>Ch 9.1, 9.2 (without analysis) [2]</td>
</tr>
</tbody>
</table>

**TutorialsDuniya.com**

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
8 Sorting in Linear Time

We have now introduced several algorithms that can sort \( n \) numbers in \( O(n \lg n) \) time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of \( n \) input numbers that causes the algorithm to run in \( \Omega(n \lg n) \) time.

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms comparison sorts. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we shall prove that any comparison sort must make \( \Omega(n \lg n) \) comparisons in the worst case to sort \( n \) elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time. Of course, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the \( \Omega(n \lg n) \) lower bound does not apply to them.

8.1 Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence \( \langle a_1, a_2, \ldots, a_n \rangle \). That is, given two elements \( a_i \) and \( a_j \), we perform one of the tests \( a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j, \) or \( a_i > a_j \) to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all the input elements are distinct. Given this assumption, comparisons of the form \( a_i = a_j \) are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons \( a_i \leq a_j, a_i \geq a_j, a_i > a_j, \) and \( a_i < a_j \) are all equivalent in that
they yield identical information about the relative order of $a_i$ and $a_j$. We therefore assume that all comparisons have the form $a_i \leq a_j$.

The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

In a decision tree, we annotate each internal node by $i:j$ for some $i$ and $j$ in the range $1 \leq i, j \leq n$, where $n$ is the number of elements in the input sequence. We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (See Section C.1 for background on permutations.) The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \leq a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on $n$ elements must appear as one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual
execution of the comparison sort. (We shall refer to such leaves as “reachable.”) Thus, we shall consider only decision trees in which each permutation appears as a reachable leaf.

**A lower bound for the worst case**

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

**Theorem 8.1**

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

**Proof** From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height $h$ with $l$ reachable leaves corresponding to a comparison sort on $n$ elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height $h$ has no more than $2^h$ leaves, we have $n! \leq l \leq 2^h$, which, by taking logarithms, implies

$$h \geq \lg(n!) \quad (\text{since the } \lg \text{ function is monotonically increasing})$$

$$= \Omega(n \lg n) \quad (\text{by equation (3.19))} .$$

**Corollary 8.2**

Heapsort and merge sort are asymptotically optimal comparison sorts.

**Proof** The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1.

**Exercises**

**8.1-1**

What is the smallest possible depth of a leaf in a decision tree for a comparison sort?
8.1-2
Obtain asymptotically tight bounds on \( \lg(n!) \) without using Stirling’s approximation. Instead, evaluate the summation \( \sum_{k=1}^{n} \lg k \) using techniques from Section A.2.

8.1-3
Show that there is no comparison sort whose running time is linear for at least half of the \( n! \) inputs of length \( n \). What about a fraction of \( 1/n \) of the inputs of length \( n \)? What about a fraction \( 1/2^n \)?

8.1-4
Suppose that you are given a sequence of \( n \) elements to sort. The input sequence consists of \( n/k \) subsequences, each containing \( k \) elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length \( n \) is to sort the \( k \) elements in each of the \( n/k \) subsequences. Show an \( \Omega(n \lg k) \) lower bound on the number of comparisons needed to solve this variant of the sorting problem. (\textit{Hint:} It is not rigorous to simply combine the lower bounds for the individual subsequences.)

8.2 Counting sort

\textit{Counting sort} assumes that each of the \( n \) input elements is an integer in the range 0 to \( k \), for some integer \( k \). When \( k = O(n) \), the sort runs in \( \Theta(n) \) time.

Counting sort determines, for each input element \( x \), the number of elements less than \( x \). It uses this information to place element \( x \) directly into its position in the output array. For example, if 17 elements are less than \( x \), then \( x \) belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

In the code for counting sort, we assume that the input is an array \( A[1..n] \), and thus \( A.length = n \). We require two other arrays: the array \( B[1..n] \) holds the sorted output, and the array \( C[0..k] \) provides temporary working storage.
8.2 Counting sort

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of $A$ is a nonnegative integer no larger than $k = 5$. (a) The array $A$ and the auxiliary array $C$ after line 5. (b) The array $C$ after line 8. (c)–(e) The output array $B$ and the auxiliary array $C$ after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array $B$ have been filled in. (f) The final sorted output array $B$.

COUNTING-SORT($A$, $B$, $k$)

1. let $C[0..k]$ be a new array
2. for $i = 0$ to $k$
3. $C[i] = 0$
4. for $j = 1$ to $A.length$
6. // $C[i]$ now contains the number of elements equal to $i$.
7. for $i = 1$ to $k$
8. $C[i] = C[i] + C[i – 1]$
9. // $C[i]$ now contains the number of elements less than or equal to $i$.
10. for $j = A.length$ downto 1

Figure 8.2 illustrates counting sort. After the for loop of lines 2–3 initializes the array $C$ to all zeros, the for loop of lines 4–5 inspects each input element. If the value of an input element is $i$, we increment $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to $i$ for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to $i$ by keeping a running sum of the array $C$. 

For any query, contact us at help@tutorialsduniya.com
Finally, the `for` loop of lines 10–12 places each element `A[j]` into its correct sorted position in the output array `B`. If all `n` elements are distinct, then when we first enter line 10, for each `A[j]`, the value `C[A[j]]` is the correct final position of `A[j]` in the output array, since there are `C[A[j]]` elements less than or equal to `A[j]`. Because the elements might not be distinct, we decrement `C[A[j]]` each time we place a value `A[j]` into the `B` array. Decrementing `C[A[j]]` causes the next input element with a value equal to `A[j]`, if one exists, to go to the position immediately before `A[j]` in the output array.

How much time does counting sort require? The `for` loop of lines 2–3 takes time \( \Theta(k) \), the `for` loop of lines 4–5 takes time \( \Theta(n) \), the `for` loop of lines 7–8 takes time \( \Theta(k) \), and the `for` loop of lines 10–12 takes time \( \Theta(n) \). Thus, the overall time is \( \Theta(k + n) \). In practice, we usually use counting sort when we have \( k = O(n) \), in which case the running time is \( \Theta(n) \).

Counting sort beats the lower bound of \( \Omega(n \lg n) \) proved in Section 8.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array. The \( \Omega(n \lg n) \) lower bound for sorting does not apply when we depart from the comparison sort model.

An important property of counting sort is that it is **stable**: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort’s stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

**Exercises**

8.2-1
Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array `A = \{6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2\}`.

8.2-2
Prove that COUNTING-SORT is stable.

8.2-3
Suppose that we were to rewrite the `for` loop header in line 10 of the COUNTING-SORT as

10  `for j = 1 to A.length`

Show that the algorithm still works properly. Is the modified algorithm stable?
8.2-4
Describe an algorithm that, given \( n \) integers in the range 0 to \( k \), preprocesses its input and then answers any query about how many of the \( n \) integers fall into a range \([a . . b]\) in \( O(1)\) time. Your algorithm should use \( \Theta(n + k) \) preprocessing time.

8.3 Radix sort

**Radix sort** is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically “programmed” to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A \( d \)-digit number would then occupy a field of \( d \) columns. Since the card sorter can look at only one column at a time, the problem of sorting \( n \) cards on a \( d \)-digit number requires a sorting algorithm.

Intuitively, you might sort numbers on their most significant digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-5.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the least significant digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all \( d \) digits. Remarkably, at that point the cards are fully sorted on the \( d \)-digit number. Thus, only \( d \) passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a “deck” of seven 3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator has to be wary about not changing the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.
In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the \( n \)-element array \( A \) has \( d \) digits, where digit 1 is the lowest-order digit and digit \( d \) is the highest-order digit.

```
RADI X-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

**Lemma 8.3**

Given \( n \) \( d \)-digit numbers in which each digit can take on up to \( k \) possible values, \( \text{RADI X-SORT} \) correctly sorts these numbers in \( \Theta(d(n + k)) \) time if the stable sort it uses takes \( \Theta(n + k) \) time.

**Proof**  The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to \( k - 1 \) (so that it can take on \( k \) possible values), and \( k \) is not too large, counting sort is the obvious choice. Each pass over \( n \) \( d \)-digit numbers then takes time \( \Theta(n + k) \). There are \( d \) passes, and so the total time for radix sort is \( \Theta(d(n + k)) \).

When \( d \) is constant and \( k = O(n) \), we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.
Lemma 8.4
Given \( n \) \( b \)-bit numbers and any positive integer \( r \leq b \), RADI X-SORT correctly sorts these numbers in \( \Theta((b/r)(n + 2^r)) \) time if the stable sort it uses takes \( \Theta(n + k) \) time for inputs in the range 0 to \( k \).

Proof  For a value \( r \leq b \), we view each key as having \( d = \lfloor b/r \rfloor \) digits of \( r \) bits each. Each digit is an integer in the range 0 to \( 2^r - 1 \), so that we can use counting sort with \( k = 2^r - 1 \). (For example, we can view a 32-bit word as having four 8-bit digits, so that \( b = 32 \), \( r = 8 \), \( k = 2^r - 1 = 255 \), and \( d = b/r = 4 \).) Each pass of counting sort takes time \( \Theta(n + k) = \Theta(n + 2^r) \) and there are \( d \) passes, for a total running time of \( \Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r)) \).

For given values of \( n \) and \( b \), we wish to choose the value of \( r \), with \( r \leq b \), that minimizes the expression \( (b/r)(n + 2^r) \). If \( b < \lfloor \lg n \rfloor \), then for any value of \( r \leq b \), we have that \( (n + 2^r) = \Theta(n) \). Thus, choosing \( r = b \) yields a running time of \( (b/b)(n + 2^b) = \Theta(n) \), which is asymptotically optimal. If \( b \geq \lfloor \lg n \rfloor \), then choosing \( r = \lfloor \lg n \rfloor \) gives the best time to within a constant factor, which we can see as follows. Choosing \( r = \lfloor \lg n \rfloor \) yields a running time of \( \Theta(bn/\lg n) \). As we increase \( r \) above \( \lfloor \lg n \rfloor \), the \( 2^r \) term in the numerator increases faster than the \( r \) term in the denominator, and so increasing \( r \) above \( \lfloor \lg n \rfloor \) yields a running time of \( \Omega(bn/\lg n) \). If instead we were to decrease \( r \) below \( \lfloor \lg n \rfloor \), then the \( b/r \) term increases and the \( n + 2^r \) term remains at \( \Theta(n) \).

Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If \( b = O(\lg n) \), as is often the case, and we choose \( r \approx \lg n \), then radix sort’s running time is \( \Theta(n) \), which appears to be better than quicksort’s expected running time of \( \Theta(n \lg n) \). The constant factors hidden in the \( \Theta \)-notation differ, however. Although radix sort may make fewer passes than quicksort over the \( n \) keys, each pass of radix sort may take significantly longer. Which sorting algorithm we prefer depends on the characteristics of the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the \( \Theta(n \lg n) \)-time comparison sorts do. Thus, when primary memory storage is at a premium, we might prefer an in-place algorithm such as quicksort.

Exercises

8.3-1
Using Figure 8.3 as a model, illustrate the operation of RADI X-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.
Chapter 8  Sorting in Linear Time

8.3-2
Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

8.3-3
Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

8.3-4
Show how to sort \( n \) integers in the range 0 to \( n^3 - 1 \) in \( O(n) \) time.

8.3-5  *
In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort \( d \)-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

8.4  Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of \( O(n) \). Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval \([0, 1]\). (See Section C.2 for a definition of uniform distribution.)

Bucket sort divides the interval \([0, 1]\) into \( n \) equal-sized subintervals, or buckets, and then distributes the \( n \) input numbers into the buckets. Since the inputs are uniformly and independently distributed over \([0, 1]\), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Our code for bucket sort assumes that the input is an \( n \)-element array \( A \) and that each element \( A[i] \) in the array satisfies \( 0 \leq A[i] < 1 \). The code requires an auxiliary array \( B[0 \ldots n - 1] \) of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.)
8.4 Bucket sort

Figure 8.4 The operation of BUCKET-SORT for \( n = 10 \). (a) The input array \( A[1..10] \). (b) The array \( B[0..9] \) of sorted lists (buckets) after line 8 of the algorithm. Bucket \( i \) holds values in the half-open interval \([i/10, (i + 1)/10)\). The sorted output consists of a concatenation in order of the lists \( B[0], B[1], \ldots, B[9] \).

**BUCKET-SORT(A)**

1. let \( B[0..n-1] \) be a new array
2. \( n = A.length \)
3. for \( i = 0 \) to \( n-1 \)
4. make \( B[i] \) an empty list
5. for \( i = 1 \) to \( n \)
6. insert \( A[i] \) into list \( B[\lfloor nA[i]\rfloor] \)
7. for \( i = 0 \) to \( n-1 \)
8. sort list \( B[i] \) with insertion sort
9. concatenate the lists \( B[0], B[1], \ldots, B[n-1] \) together in order

Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

To see that this algorithm works, consider two elements \( A[i] \) and \( A[j] \). Assume without loss of generality that \( A[i] \leq A[j] \). Since \( \lfloor nA[i]\rfloor \leq \lfloor nA[j]\rfloor \), either element \( A[i] \) goes into the same bucket as \( A[j] \) or it goes into a bucket with a lower index. If \( A[i] \) and \( A[j] \) go into the same bucket, then the for loop of lines 7–8 puts them into the proper order. If \( A[i] \) and \( A[j] \) go into different buckets, then line 9 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that all lines except line 8 take \( O(n) \) time in the worst case. We need to analyze the total time taken by the \( n \) calls to insertion sort in line 8.
To analyze the cost of the calls to insertion sort, let \( n_i \) be the random variable denoting the number of elements placed in bucket \( B[i] \). Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

\[
T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)
\]

We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation, we have

\[
\mathbb{E}[T(n)] = \mathbb{E} \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right]
\]

\[
= \Theta(n) + \sum_{i=0}^{n-1} \mathbb{E}[O(n_i^2)] \quad \text{(by linearity of expectation)}
\]

\[
= \Theta(n) + \sum_{i=0}^{n-1} O\left( \mathbb{E}[n_i^2] \right) \quad \text{(by equation (C.22))}
\]  

(8.1)

We claim that

\[
\mathbb{E}[n_i^2] = 2 - 1/n
\]

(8.2)

for \( i = 0, 1, \ldots, n - 1 \). It is no surprise that each bucket \( i \) has the same value of \( \mathbb{E}[n_i^2] \), since each value in the input array \( A \) is equally likely to fall in any bucket. To prove equation (8.2), we define indicator random variables

\[
X_{ij} = 1 \{ A[j] \text{ falls in bucket } i \}
\]

for \( i = 0, 1, \ldots, n - 1 \) and \( j = 1, 2, \ldots, n \). Thus,

\[
n_i = \sum_{j=1}^{n} X_{ij}
\]

To compute \( \mathbb{E}[n_i^2] \), we expand the square and regroup terms:

For any query, contact us at help@tutorialsduniya.com
8.4 Bucket sort

\[ E \left[ n_i^2 \right] = E \left[ \left( \sum_{j=1}^{n} X_{ij} \right)^2 \right] \]

\[ = E \left[ \sum_{j=1}^{n} \sum_{k=1}^{n} X_{ij} X_{ik} \right] \]

\[ = E \left[ \sum_{j=1}^{n} X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n} X_{ij} X_{ik} \right] \]

\[ = \sum_{j=1}^{n} E \left[ X_{ij}^2 \right] + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n} E \left[ X_{ij} X_{ik} \right] , \quad (8.3) \]

where the last line follows by linearity of expectation. We evaluate the two summations separately. Indicator random variable \( X_{ij} \) is 1 with probability \( \frac{1}{n} \) and 0 otherwise, and therefore

\[ E \left[ X_{ij}^2 \right] \]

\[ = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left( 1 - \frac{1}{n} \right) \]

\[ = \frac{1}{n} . \]

When \( k \neq j \), the variables \( X_{ij} \) and \( X_{ik} \) are independent, and hence

\[ E \left[ X_{ij} X_{ik} \right] = E \left[ X_{ij} \right] E \left[ X_{ik} \right] \]

\[ = \frac{1}{n} \cdot \frac{1}{n} \]

\[ = \frac{1}{n^2} . \]

Substituting these two expected values in equation (8.3), we obtain

\[ E \left[ n_i^2 \right] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n} \frac{1}{n^2} \]

\[ = n \cdot \frac{1}{n} + n(n - 1) \cdot \frac{1}{n^2} \]

\[ = 1 + \frac{n - 1}{n} \]

\[ = 2 - \frac{1}{n} , \]

which proves equation (8.2).
Using this expected value in equation (8.1), we conclude that the average-case running time for bucket sort is \( \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n) \).

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort will run in linear time.

**Exercises**

8.4-1 Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array \( A = (0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42) \).

8.4-2 Explain why the worst-case running time for bucket sort is \( \Theta(n^2) \). What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time \( O(n \log n) \)?

8.4-3 Let \( X \) be a random variable that is equal to the number of heads in two flips of a fair coin. What is \( E[X^2] \)? What is \( E^2[X] \)?

8.4-4 Using \( n \) points in the unit circle, \( p_i = (x_i, y_i) \), such that \( 0 < x_i^2 + y_i^2 \leq 1 \) for \( i = 1, 2, \ldots, n \). Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of \( \Theta(n) \) to sort the \( n \) points by their distances \( d_i = \sqrt{x_i^2 + y_i^2} \) from the origin. (Hint: Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

8.4-5 A probability distribution function \( P(x) \) for a random variable \( X \) is defined by \( P(x) = \Pr\{X \leq x\} \). Suppose that we draw a list of \( n \) random variables \( X_1, X_2, \ldots, X_n \) from a continuous probability distribution function \( P \) that is computable in \( O(1) \) time. Give an algorithm that sorts these numbers in linear average-case time.

For any query, contact us at help@tutorialsduniya.com
Problems for Chapter 8

8-1 Probabilistic lower bounds on comparison sorting
In this problem, we prove a probabilistic \( \Omega(n \lg n) \) lower bound on the running time of any deterministic or randomized comparison sort on \( n \) distinct input elements. We begin by examining a deterministic comparison sort \( A \) with decision tree \( T_A \). We assume that every permutation of \( A \)'s inputs is equally likely.

\( a. \) Suppose that each leaf of \( T_A \) is labeled with the probability that it is reached given a random input. Prove that exactly \( n! \) leaves are labeled \( \frac{1}{n!} \) and that the rest are labeled \( 0 \).

\( b. \) Let \( D(T) \) denote the external path length of a decision tree \( T \); that is, \( D(T) \) is the sum of the depths of all the leaves of \( T \). Let \( T \) be a decision tree with \( k > 1 \) leaves, and let \( LT \) and \( RT \) be the left and right subtrees of \( T \). Show that \( D(T) = D(LT) + D(RT) + k \).

\( c. \) Let \( d(k) \) be the minimum value of \( D(T) \) over all decision trees \( T \) with \( k > 1 \) leaves. Show that \( d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\} \). (Hint: Consider a decision tree \( T \) with \( k \) leaves that achieves the minimum. Let \( i_0 \) be the number of leaves in \( LT \) and \( k - i_0 \) the number of leaves in \( RT \).)

\( d. \) Prove that for a given value of \( k > 1 \) and \( i \) in the range \( 1 \leq i \leq k - 1 \), the function \( i \lg i + (k - i) \lg(k - i) \) is minimized at \( i = k/2 \). Conclude that \( d(k) = \Omega(k \lg k) \).

\( e. \) Prove that \( D(T_A) = \Omega(n! \lg(n!)) \), and conclude that the average-case time to sort \( n \) elements is \( \Omega(n \lg n) \).

Now, consider a randomized comparison sort \( B \). We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and “randomization” nodes. A randomization node models a random choice of the form \( \text{RANDOM}(1, r) \) made by algorithm \( B \); the node has \( r \) children, each of which is equally likely to be chosen during an execution of the algorithm.

\( f. \) Show that for any randomized comparison sort \( B \), there exists a deterministic comparison sort \( A \) whose expected number of comparisons is no more than those made by \( B \).
8-2 Sorting in place in linear time
Suppose that we have an array of \( n \) data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in \( O(n) \) time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

\( a. \) Give an algorithm that satisfies criteria 1 and 2 above.

\( b. \) Give an algorithm that satisfies criteria 1 and 3 above.

\( c. \) Give an algorithm that satisfies criteria 2 and 3 above.

\( d. \) Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts \( n \) records with \( b \)-bit keys in \( O(bn) \) time? Explain how or why not.

\( e. \) Suppose that the \( n \) records have keys in the range from 1 to \( k \). Show how to modify counting sort so that it sorts the records in place in \( O(n + k) \) time. You may use \( O(k) \) storage outside the input array. Is your algorithm stable? (Hint: How would you do it for \( k = 3 \)?)

8-3 Sorting variable-length items
\( a. \) You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is \( n \). Show how to sort the array in \( O(n) \) time.

\( b. \) You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is \( n \). Show how to sort the strings in \( O(n) \) time.

(Note that the desired order here is the standard alphabetical order; for example, \( a < ab < b \).)

8-4 Water jugs
Suppose that you are given \( n \) red and \( n \) blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.
Problems for Chapter 8

Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an $n$-element array $A$ $k$-sorted if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$ 

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers 1, 2, \ldots, 10 that is 2-sorted, but not sorted.

c. Prove that an $n$-element array is $k$-sorted if and only if $A[i] \leq A[i + k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that $k$-sorts an $n$-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a $k$-sorted array, when $k$ is a constant.

e. Show that we can sort a $k$-sorted array of length $n$ in $O(n \lg k)$ time. (Hint: Use the solution to Exercise 6.5-9.)

f. Show that when $k$ is a constant, $k$-sorting an $n$-element array requires $\Omega(n \lg n)$ time. (Hint: Use the solution to the previous part along with the lower bound on comparison sorts.)
8-6 Lower bound on merging sorted lists
The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will prove a lower bound of \( 2n - 1 \) on the worst-case number of comparisons required to merge two sorted lists, each containing \( n \) items.

First we will show a lower bound of \( 2n - o(n) \) comparisons by using a decision tree.

\( a \). Given \( 2n \) numbers, compute the number of possible ways to divide them into two sorted lists, each with \( n \) numbers.

\( b \). Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least \( 2n - o(n) \) comparisons.

Now we will show a slightly tighter \( 2n - 1 \) bound.

\( c \). Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

\( d \). Use your answer to the previous part to show a lower bound of \( 2n - 1 \) comparisons for merging two sorted lists.

8-7 The 0-1 sorting lemma and columnsort
A compare-exchange operation on two array elements \( A[i] \) and \( A[j] \), where \( i < j \), has the form

\[
\text{COMPARE-EXCHANGE}(A, i, j)
\]

1. \( \text{if } A[i] > A[j] \)
2. \( \text{exchange } A[i] \text{ with } A[j] \)

After the compare-exchange operation, we know that \( A[i] \leq A[j] \).

An oblivious compare-exchange algorithm operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, here is insertion sort expressed as an oblivious compare-exchange algorithm:

\[
\text{INSERTION-SORT}(A)
\]

1. \( \text{for } j = 2 \text{ to } A\.length \)
2. \( \text{for } i = j - 1 \text{ downto } 1 \)
3. \( \text{COMPARE-EXCHANGE}(A, i, i + 1) \)
The 0-1 sorting lemma provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array \( A[1..n] \). Let \( A[p] \) be the smallest value in \( A \) that algorithm X puts into the wrong location, and let \( A[q] \) be the value that algorithm X moves to the location into which \( A[p] \) should have gone. Define an array \( B[1..n] \) of 0s and 1s as follows:

\[
B[i] = \begin{cases} 
0 & \text{if } A[i] \leq A[p], \\
1 & \text{if } A[i] > A[p].
\end{cases}
\]

**a.** Argue that \( A[q] > A[p] \), so that \( B[p] = 0 \) and \( B[q] = 1 \).

**b.** To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array \( B \) correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, columnsort, works on a rectangular array of \( n \) elements. The array has \( r \) rows and \( s \) columns (so that \( n = rs \)), subject to three restrictions:

- \( r \) must be even,
- \( s \) must be a divisor of \( r \), and
- \( r \geq 2s^2 \).

When columnsort completes, the array is sorted in column-major order: reading down the columns, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of \( n \). The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to \( r \) rows and \( s \) columns. In other words, turn the leftmost column into the top \( r/s \) rows, in order; turn the next column into the next \( r/s \) rows, in order; and so on.
3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and
shift the bottom half of each column into the top half of the next column to the
right. Leave the top half of the leftmost column empty. Shift the bottom half
of the last column into the top half of a new rightmost column, and leave the
bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

Figure 8.5 shows an example of the steps of columnsort with \( r = 6 \) and \( s = 3 \).
(Even though this example violates the requirement that \( r \geq 2s^2 \), it happens to
work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algo-
rithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will
use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies
because we can treat columnsort as an oblivious compare-exchange algorithm. A
couples of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is **clean** if we know that it contains either all 0s or all 1s. Otherwise, the area might contain mixed 0s and 1s, and it is **dirty**. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with $r$ rows and $s$ columns.

**d.** Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $s$ dirty rows between them.

**e.** Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most $s^2$ elements in the middle.

**f.** Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

**g.** Now suppose that $s$ does not divide $r$. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. How large must $r$ be, compared with $s$, for columnsort to correctly sort when $s$ does not divide $r$?

**h.** Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when $s$ does not divide $r$, and prove that with your change, columnsort correctly sorts.
9 Medians and Order Statistics

The \( i \)th order statistic of a set of \( n \) elements is the \( i \)th smallest element. For example, the minimum of a set of elements is the first order statistic (\( i = 1 \)), and the maximum is the \( n \)th order statistic (\( i = n \)). A median, informally, is the “halfway point” of the set. When \( n \) is odd, the median is unique, occurring at \( i = (n + 1)/2 \). When \( n \) is even, there are two medians, occurring at \( i = n/2 \) and \( i = n/2 + 1 \). Thus, regardless of the parity of \( n \), medians occur at \( i = \lfloor (n + 1)/2 \rfloor \) (the lower median) and \( i = \lceil (n + 1)/2 \rceil \) (the upper median). For simplicity in this text, however, we consistently use the phrase “the median” to refer to the lower median.

This chapter addresses the problem of selecting the \( i \)th order statistic from a set of \( n \) distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the selection problem as follows:

**Input:** A set \( A \) of \( n \) (distinct) numbers and an integer \( i \), with \( 1 \leq i \leq n \).

**Output:** The element \( x \in A \) that is larger than exactly \( i - 1 \) other elements of \( A \).

We can solve the selection problem in \( O(n \lg n) \) time, since we can sort the numbers using heapsort or merge sort and then simply index the \( i \)th element in the output array. This chapter presents faster algorithms.

In Section 9.1, we examine the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which we investigate in the subsequent two sections. Section 9.2 analyzes a practical randomized algorithm that achieves an \( O(n) \) expected running time, assuming distinct elements. Section 9.3 contains an algorithm of more theoretical interest that achieves the \( O(n) \) running time in the worst case.
9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of \( n \) elements? We can easily obtain an upper bound of \( n - 1 \) comparisons: examine each element of the set in turn and keep track of the smallest element seen so far. In the following procedure, we assume that the set resides in array \( A \), where \( A.length = n \).

\begin{verbatim}
MINIMUM(A)
1  \textit{min} = A[1]
2  \textbf{for} i = 2 \textbf{to} A.length
3      \textbf{if} \textit{min} > A[i]
4          \textit{min} = A[i]
5  \textbf{return} \textit{min}
\end{verbatim}

We can, of course, find the maximum with \( n - 1 \) comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of \( n - 1 \) comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. Observing that every element except the winner must lose at least one match, we conclude that \( n - 1 \) comparisons are necessary to determine the minimum. Hence, the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of \( n \) elements. For example, a graphics program may need to scale a set of \((x, y)\) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

At this point, it should be obvious how to determine both the minimum and the maximum of \( n \) elements using \( \Theta(n) \) comparisons, which is asymptotically optimal: simply find the minimum and maximum independently, using \( n - 1 \) comparisons for each, for a total of \( 2n - 2 \) comparisons.

In fact, we can find both the minimum and the maximum using at most \( 3 \left\lceil \frac{n}{2} \right\rceil \) comparisons. We do so by maintaining both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element,
9.2 Selection in expected linear time

we process elements in pairs. We compare pairs of elements from the input first with each other, and then we compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How we set up initial values for the current minimum and maximum depends on whether \( n \) is odd or even. If \( n \) is odd, we set both the minimum and maximum to the value of the first element, and then we process the rest of the elements in pairs. If \( n \) is even, we perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd \( n \).

Let us analyze the total number of comparisons. If \( n \) is odd, then we perform \( 3 \lfloor n/2 \rfloor \) comparisons. If \( n \) is even, we perform 1 initial comparison followed by \( 3(n - 2)/2 \) comparisons, for a total of \( 3n/2 - 2 \). Thus, in either case, the total number of comparisons is at most \( 3 \lfloor n/2 \rfloor \).

Exercises

9.1-1
Show that the second smallest of \( n \) elements can be found with \( n + \lfloor \lg n \rfloor - 2 \) comparisons in the worst case. (Hint: Also find the smallest element.)

9.1-2 ★
Prove the lower bound of \( \lceil 3n/2 \rceil - 2 \) comparisons in the worst case to find both the maximum and minimum of \( n \) numbers. (Hint: Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

9.2 Selection in expected linear time

The general selection problem appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: \( \Theta(n) \). In this section, we present a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. As in quicksort, we partition the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of \( \Theta(n \lg n) \), the expected running time of RANDOMIZED-SELECT is \( \Theta(n) \), assuming that the elements are distinct.
RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for RANDOMIZED-SELECT returns the $i$th smallest element of the array $A[p..r]$.

**RANDOMIZED-SELECT**$(A, p, r, i)$

1. if $p == r$
2. return $A[p]$
3. $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
4. $k = q - p + 1$
5. if $i == k$ // the pivot value is the answer
6. return $A[q]$
7. elseif $i < k$
8. return RANDOMIZED-SELECT$(A, p, q - 1, i)$
9. else return RANDOMIZED-SELECT$(A, q + 1, r, i - k)$

The RANDOMIZED-SELECT procedure works as follows. Line 1 checks for the base case of the recursion, in which the subarray $A[p..r]$ consists of just one element. In this case, $i$ must equal 1, and we simply return $A[p]$ in line 2 as the $i$th smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q - 1]$ and $A[q + 1..r]$ such that each element of $A[p..q - 1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q + 1..r]$. As in quicksort, we will refer to $A[q]$ as the pivot element. Line 4 computes the number $k$ of elements in the subarray $A[p..q]$, that is, the number of elements in the low side of the partition, plus one for the pivot element. Line 5 then checks whether $A[q]$ is the $i$th smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p..q - 1]$ and $A[q + 1..r]$ the $i$th smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know $k$ values that are smaller than the $i$th smallest element of $A[p..r]$—namely, the elements of $A[p..q]$—the desired element is the $(i - k)$th smallest element of $A[q + 1..r]$, which line 9 finds recursively. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element, and partitioning takes $\Theta(n)$ time. We will see that
9.2 Selection in expected linear time

the algorithm has a linear expected running time, though, and because it is randomized, no particular input elicits the worst-case behavior.

To analyze the expected running time of \textsc{Randomized-Select}, we let the running time on an input array \( A[p..r] \) of \( n \) elements be a random variable that we denote by \( T(n) \), and we obtain an upper bound on \( E[T(n)] \) as follows. The procedure \textsc{Randomized-Partition} is equally likely to return any element as the pivot. Therefore, for each \( k \) such that \( 1 \leq k \leq n \), the subarray \( A[p..q] \) has \( k \) elements (all less than or equal to the pivot) with probability \( 1/n \). For \( k = 1, 2, \ldots, n \), we define indicator random variables \( X_k \) where

\[
X_k = \begin{cases} 
1 & \text{the subarray } A[p..q] \text{ has exactly } k \text{ elements} \\
0 & \text{otherwise}
\end{cases}
\]

and so, assuming that the elements are distinct, we have

\[
E[X_k] = 1/n \quad \text{(9.1)}
\]

When we call \textsc{Randomized-Select} and choose \( A[q] \) as the pivot element, we do not know, a priori, if we will terminate immediately with the correct answer, recurse on the subarray \( A[p..q-1] \), or recurse on the subarray \( A[q+1..r] \). This decision depends on where the \( i \)th smallest element falls relative to \( A[q] \). Assuming that \( T(n) \) is monotonically increasing, we can upper-bound the time needed for the recursive call by the time needed for the recursive call on the largest possible input. In other words, to obtain an upper bound, we assume that the \( i \)th element is always on the side of the partition with the greater number of elements. For a given call of \textsc{Randomized-Select}, the indicator random variable \( X_k \) has the value 1 for exactly one value of \( k \), and it is 0 for all other \( k \). When \( X_k = 1 \), the two subarrays on which we might recurse have sizes \( k-1 \) and \( n-k \). Hence, we have the recurrence

\[
T(n) \leq \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))
\]

\[
= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)
\]
Taking expected values, we have

\[
E[T(n)] 
\leq E \left[ \sum_{k=1}^{n} X_k \cdot T(\max(k - 1, n - k)) + O(n) \right] 
\]

\[
= \sum_{k=1}^{n} E[X_k \cdot T(\max(k - 1, n - k))] + O(n) \quad \text{(by linearity of expectation)} 
\]

\[
= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k - 1, n - k))] + O(n) \quad \text{(by equation (C.24))} 
\]

\[
= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k - 1, n - k))] + O(n) \quad \text{(by equation (9.1))} . 
\]

In order to apply equation (C.24), we rely on \(X_k\) and \(T(\max(k - 1, n - k))\) being independent random variables. Exercise 9.2-2 asks you to justify this assertion.

Let us consider the expression \(\max(k - 1, n - k)\). We have

\[
\max(k - 1, n - k) = \begin{cases} 
  k - 1 & \text{if } k > \lfloor n/2 \rfloor , \\
  n - k & \text{if } k \leq \lfloor n/2 \rfloor . 
\end{cases} 
\]

If \(n\) is even, each term from \(T(\lfloor n/2 \rfloor)\) up to \(T(n - 1)\) appears exactly twice in the summation, and if \(n\) is odd, all these terms appear twice and \(T(\lfloor n/2 \rfloor)\) appears once. Thus, we have

\[
E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n) . 
\]

We show that \(E[T(n)] = O(n)\) by substitution. Assume that \(E[T(n)] \leq cn\) for some constant \(c\) that satisfies the initial conditions of the recurrence. We assume that \(T(n) = O(1)\) for \(n\) less than some constant; we shall pick this constant later.

We also pick a constant \(a\) such that the function described by the \(O(n)\) term above (which describes the non-recursive component of the running time of the algorithm) is bounded from above by \(an\) for all \(n > 0\). Using this inductive hypothesis, we have

\[
E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an 
\]

\[
= \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an 
\]
In order to complete the proof, we need to show that for sufficiently large $n$, this last expression is at most $cn$ or, equivalently, that $cn/4 - c/2 - an \geq 0$. If we add $c/2$ to both sides and factor out $n$, we get $n(c/4 - a) \geq c/2$. As long as we choose the constant $c$ so that $c/4 - a > 0$, i.e., $c > 4a$, we can divide both sides by $c/4 - a$, giving

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

Thus, if we assume that $T(n) = O(1)$ for $n < 2c/(c - 4a)$, then $E[T(n)] = O(n)$. We conclude that we can find any order statistic, and in particular the median, in expected linear time, assuming that the elements are distinct.

**Exercises**

**9.2-1**
Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

**9.2-2**
Argue that the indicator random variable $X_k$ and the value $T(\max(k - 1, n - k))$ are independent.

**9.2-3**
Write an iterative version of RANDOMIZED-SELECT.
9.2-4
Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.
Problems

9-1 Largest $i$ numbers in sorted order
Given a set of $n$ numbers, we wish to find the $i$ largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of $n$ and $i$.

a. Sort the numbers, and list the $i$ largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX $i$ times.

c. Use an order-statistic algorithm to find the $i$th largest number, partition around that number, and sort the $i$ largest numbers.
9-2 Weighted median

For \( n \) distinct elements \( x_1, x_2, \ldots, x_n \) with positive weights \( w_1, w_2, \ldots, w_n \) such that \( \sum_{i=1}^{n} w_i = 1 \), the weighted (lower) median is the element \( x_k \) satisfying

\[
\sum_{x_i < x_k} w_i < \frac{1}{2}
\]

and

\[
\sum_{x_i > x_k} w_i \leq \frac{1}{2}.
\]

For example, if the elements are \( 0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2 \) and each element equals its weight (that is, \( w_i = x_i \) for \( i = 1, 2, \ldots, 7 \)), then the median is 0.1, but the weighted median is 0.2.

\( a. \) Argue that the median of \( x_1, x_2, \ldots, x_n \) is the weighted median of the \( x_i \) with weights \( w_i = 1/n \) for \( i = 1, 2, \ldots, n \).

\( b. \) Show how to compute the weighted median of \( n \) elements in \( O(n \lg n) \) worst-case time using sorting.

\( c. \) Show how to compute the weighted median in \( \Theta(n) \) worst-case time using a linear-time median algorithm such as Select from Section 9.3.

The \textit{post-office location problem} is defined as follows. We are given \( n \) points \( p_1, p_2, \ldots, p_n \) with associated weights \( w_1, w_2, \ldots, w_n \). We wish to find a point \( p \) (not necessarily one of the input points) that minimizes the sum \( \sum_{i=1}^{n} w_i d(p, p_i) \), where \( d(a, b) \) is the distance between points \( a \) and \( b \).

\( d. \) Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points \( a \) and \( b \) is \( d(a, b) = |a - b| \).

\( e. \) Find the best solution for the 2-dimensional post-office location problem, in which the points are \( (x, y) \) coordinate pairs and the distance between points \( a = (x_1, y_1) \) and \( b = (x_2, y_2) \) is the \textit{Manhattan distance} given by \( d(a, b) = |x_1 - x_2| + |y_1 - y_2| \).

9-3 Small order statistics

We showed that the worst-case number \( T(n) \) of comparisons used by Select to select the \( i \)th order statistic from \( n \) numbers satisfies \( T(n) = \Theta(n) \), but the constant hidden by the \( \Theta \)-notation is rather large. When \( i \) is small relative to \( n \), we can implement a different procedure that uses Select as a subroutine but makes fewer comparisons in the worst case.
Chapter 9  Medians and Order Statistics

a. Describe an algorithm that uses $U_i(n)$ comparisons to find the $i$th smallest of $n$ elements, where

$$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2, \\ \lfloor n/2 \rfloor + U_i(\lfloor n/2 \rfloor) + T(2i) & \text{otherwise}. \end{cases}$$

*(Hint: Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)*

b. Show that, if $i < n/2$, then $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Show that if $i$ is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

d. Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(T(2n/k) \lg k)$.

9-4 Alternative analysis of randomized selection

In this problem, we use indicator random variables to analyze the RANDOMIZED-SELECT procedure in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array $A$ as $z_1, z_2, \ldots, z_n$, where $z_i$ is the $i$th smallest element. Thus, the call RANDOMIZED-SELECT($A, 1, n, k$) returns $z_k$.

For $1 \leq i < j \leq n$, let

$$X_{ijk} = I \{ z_i \text{ is compared with } z_j \text{ sometime during the execution of the algorithm to find } z_k \}.$$

a. Give an exact expression for $E[X_{ijk}]$. *(Hint: Your expression may have different values, depending on the values of $i$, $j$, and $k$).*

b. Let $X_k$ denote the total number of comparisons between elements of array $A$ when finding $z_k$. Show that

$$E[X_k] \leq 2 \left( \sum_{i=1}^{k} \sum_{j=k}^{n} \frac{1}{j-i+1} + \sum_{j=k+1}^{n} \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right).$$

c. Show that $E[X_k] \leq 4n$.

d. Conclude that, assuming all elements of array $A$ are distinct, RANDOMIZED-SELECT runs in expected time $O(n)$.
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>Graph Algorithms</td>
<td>Ch – 3 [3]</td>
</tr>
<tr>
<td></td>
<td>Amortized Analysis</td>
<td>Ch – 17.1 [2]</td>
</tr>
</tbody>
</table>

**TutorialsDuniya.com**

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
## TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

<table>
<thead>
<tr>
<th>Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
</tr>
<tr>
<td>Programming in Java</td>
</tr>
<tr>
<td>Data Structures</td>
</tr>
<tr>
<td>Computer Networks</td>
</tr>
<tr>
<td>Android Programming</td>
</tr>
<tr>
<td>PHP Programming</td>
</tr>
<tr>
<td>JavaScript</td>
</tr>
<tr>
<td>Java Server Pages</td>
</tr>
<tr>
<td>Python</td>
</tr>
<tr>
<td>Microprocessor</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>Machine Learning</td>
</tr>
<tr>
<td>Computer System Architecture</td>
</tr>
<tr>
<td>Discrete Structures</td>
</tr>
<tr>
<td>Operating Systems</td>
</tr>
<tr>
<td>Algorithms</td>
</tr>
<tr>
<td>Database Management Systems</td>
</tr>
<tr>
<td>Software Engineering</td>
</tr>
<tr>
<td>Theory of Computation</td>
</tr>
<tr>
<td>Operational Research</td>
</tr>
<tr>
<td>System Programming</td>
</tr>
<tr>
<td>Data Mining</td>
</tr>
<tr>
<td>Computer Graphics</td>
</tr>
<tr>
<td>Data Science</td>
</tr>
</tbody>
</table>

- **Compiled Books**: [https://www.tutorialsduniya.com/compiled-books](https://www.tutorialsduniya.com/compiled-books)
- **Programs**: [https://www.tutorialsduniya.com/programs](https://www.tutorialsduniya.com/programs)
- **Question Papers**: [https://www.tutorialsduniya.com/question-papers](https://www.tutorialsduniya.com/question-papers)
- **Python Notes**: [https://www.tutorialsduniya.com/python](https://www.tutorialsduniya.com/python)
- **Java Notes**: [https://www.tutorialsduniya.com/java](https://www.tutorialsduniya.com/java)
- **JavaScript Notes**: [https://www.tutorialsduniya.com/javascript](https://www.tutorialsduniya.com/javascript)
- **JSP Notes**: [https://www.tutorialsduniya.com/jsp](https://www.tutorialsduniya.com/jsp)
- **Microprocessor Notes**: [https://www.tutorialsduniya.com/microprocessor](https://www.tutorialsduniya.com/microprocessor)
- **OR Notes**: [https://www.tutorialsduniya.com/operational-research](https://www.tutorialsduniya.com/operational-research)
Chapter 3

Graphs

Our focus in this book is on problems with a discrete flavor. Just as continuous mathematics is concerned with certain basic structures such as real numbers, vectors, and matrices, discrete mathematics has developed basic combinatorial structures that lie at the heart of the subject. One of the most fundamental and expressive of these is the graph.

The more one works with graphs, the more one tends to see them everywhere. Thus, we begin by introducing the basic definitions surrounding graphs, and list a spectrum of different algorithmic settings where graphs arise naturally. We then discuss some basic algorithmic primitives for graphs, beginning with the problem of connectivity and developing some fundamental graph search techniques.

3.1 Basic Definitions and Applications

Recall from Chapter 1 that a graph $G$ is simply a way of encoding pairwise relationships among a set of objects: it consists of a collection $V$ of nodes and a collection $E$ of edges, each of which “joins” two of the nodes. We thus represent an edge $e \in E$ as a two-element subset of $V$: $e = \{u, v\}$ for some $u, v \in V$, where we call $u$ and $v$ the ends of $e$.

Edges in a graph indicate a symmetric relationship between their ends. Often we want to encode asymmetric relationships, and for this we use the closely related notion of a directed graph. A directed graph $G'$ consists of a set of nodes $V$ and a set of directed edges $E'$. Each $e' \in E'$ is an ordered pair $(u, v)$; in other words, the roles of $u$ and $v$ are not interchangeable, and we call $u$ the tail of the edge and $v$ the head. We will also say that edge $e'$ leaves node $u$ and enters node $v$. 
Chapter 3  Graphs

When we want to emphasize that the graph we are considering is not directed, we will call it an undirected graph; by default, however, the term “graph” will mean an undirected graph. It is also worth mentioning two warnings in our use of graph terminology. First, although an edge $e$ in an undirected graph should properly be written as a set of nodes $\{u, v\}$, one will more often see it written (even in this book) in the notation used for ordered pairs: $e = (u, v)$. Second, a node in a graph is also frequently called a vertex; in this context, the two words have exactly the same meaning.

**Examples of Graphs** Graphs are very simple to define: we just take a collection of things and join some of them by edges. But at this level of abstraction, it’s hard to appreciate the typical kinds of situations in which they arise. Thus, we propose the following list of specific contexts in which graphs serve as important models. The list covers a lot of ground, and it’s not important to remember everything on it; rather, it will provide us with a lot of useful examples against which to check the basic definitions and algorithmic problems that we’ll be encountering later in the chapter. Also, in going through the list, it’s useful to digest the meaning of the nodes and the meaning of the edges in the context of the application. In some cases the nodes and edges both correspond to physical objects in the real world, in others the nodes are real objects while the edges are virtual, and in still others both nodes and edges are pure abstractions.

1. *Transportation networks.* The map of routes served by an airline carrier naturally forms a graph: the nodes are airports, and there is an edge from $u$ to $v$ if there is a nonstop flight that departs from $u$ and arrives at $v$. Described this way, the graph is directed; but in practice when there is an edge $(u, v)$, there is almost always an edge $(v, u)$, so we would not lose much by treating the airline route map as an undirected graph with edges joining pairs of airports that have nonstop flights each way. Looking at such a graph (you can generally find them depicted in the backs of inflight airline magazines), we’d quickly notice a few things: there are often a small number of hubs with a very large number of incident edges; and it’s possible to get between any two nodes in the graph via a very small number of intermediate stops.

Other transportation networks can be modeled in a similar way. For example, we could take a rail network and have a node for each terminal, and an edge joining $u$ and $v$ if there’s a section of railway track that goes between them without stopping at any intermediate terminal. The standard depiction of the subway map in a major city is a drawing of such a graph.

2. *Communication networks.* A collection of computers connected via a communication network can be naturally modeled as a graph in a few
3.1 Basic Definitions and Applications

different ways. First, we could have a node for each computer and an edge joining $u$ and $v$ if there is a direct physical link connecting them. Alternatively, for studying the large-scale structure of the Internet, people often define a node to be the set of all machines controlled by a single Internet service provider, with an edge joining $u$ and $v$ if there is a direct \textit{peering relationship} between them—roughly, an agreement to exchange data under the standard BGP protocol that governs global Internet routing. Note that this latter network is more “virtual” than the former, since the links indicate a formal agreement in addition to a physical connection.

In studying wireless networks, one typically defines a graph where the nodes are computing devices situated at locations in physical space, and there is an edge from $u$ to $v$ if $v$ is close enough to $u$ to receive a signal from it. Note that it’s often useful to view such a graph as directed, since it may be the case that $v$ can hear $u$’s signal but $u$ cannot hear $v$’s signal (if, for example, $u$ has a stronger transmitter). These graphs are also interesting from a geometric perspective, since they roughly correspond to putting down points in the plane and then joining pairs that are close together.

3. \textit{Information networks}. The World Wide Web can be naturally viewed as a directed graph, in which nodes correspond to Web pages and there is an edge from $u$ to $v$ if $u$ has a hyperlink to $v$. The directedness of the graph is crucial here; many pages, for example, link to popular news sites, but these sites clearly do not reciprocate all these links. The structure of all these hyperlinks can be used by algorithms to try inferring the most important pages on the Web, a technique employed by most current search engines.

The hypertextual structure of the Web is anticipated by a number of information networks that predate the Internet by many decades. These include the network of cross-references among articles in an encyclopedia or other reference work, and the network of bibliographic citations among scientific papers.

4. \textit{Social networks}. Given any collection of people who interact (the employees of a company, the students in a high school, or the residents of a small town), we can define a network whose nodes are people, with an edge joining $u$ and $v$ if they are friends with one another. We could have the edges mean a number of different things instead of friendship: the undirected edge $(u, v)$ could mean that $u$ and $v$ have had a romantic relationship or a financial relationship; the directed edge $(u, v)$ could mean that $u$ seeks advice from $v$, or that $u$ lists $v$ in his or her e-mail address book. One can also imagine bipartite social networks based on a
notion of affiliation: given a set $X$ of people and a set $Y$ of organizations, we could define an edge between $u \in X$ and $v \in Y$ if person $u$ belongs to organization $v$.

Networks such as this are used extensively by sociologists to study the dynamics of interaction among people. They can be used to identify the most “influential” people in a company or organization, to model trust relationships in a financial or political setting, and to track the spread of fads, rumors, jokes, diseases, and e-mail viruses.

5. Dependency networks. It is natural to define directed graphs that capture the interdependencies among a collection of objects. For example, given the list of courses offered by a college or university, we could have a node for each course and an edge from $u$ to $v$ if $u$ is a prerequisite for $v$. Given a list of functions or modules in a large software system, we could have a node for each function and an edge from $u$ to $v$ if $u$ invokes $v$ by a function call. Or given a set of species in an ecosystem, we could define a graph—a food web—in which the nodes are the different species and there is an edge from $u$ to $v$ if $u$ consumes $v$.

This is far from a complete list, too far to even begin tabulating its omissions. It is meant simply to suggest some examples that are useful to keep in mind when we start thinking about graphs in an algorithmic context.

**Paths and Connectivity**  One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges. In the examples just listed, such a traversal could correspond to a user browsing Web pages by following hyperlinks; a rumor passing by word of mouth from you to someone halfway around the world; or an airline passenger traveling from San Francisco to Rome on a sequence of flights.

With this notion in mind, we define a path in an undirected graph $G = (V, E)$ to be a sequence $P$ of nodes $v_1, v_2, \ldots, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $G$. $P$ is often called a path from $v_1$ to $v_k$, or a $v_1$-$v_k$ path. For example, the nodes $4, 2, 1, 7, 8$ form a path in Figure 3.1. A path is called simple if all its vertices are distinct from one another. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $k > 2$, the first $k - 1$ nodes are all distinct, and $v_1 = v_k$—in other words, the sequence of nodes “cycles back” to where it began. All of these definitions carry over naturally to directed graphs, with the following change: in a directed path or cycle, each pair of consecutive nodes has the property that $(v_i, v_{i+1})$ is an edge. In other words, the sequence of nodes in the path or cycle must respect the directionality of edges.

We say that an undirected graph is connected if, for every pair of nodes $u$ and $v$, there is a path from $u$ to $v$. Choosing how to define connectivity of a
3.1 Basic Definitions and Applications

Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

directed graph is a bit more subtle, since it’s possible for \( u \) to have a path to \( v \) while \( v \) has no path to \( u \). We say that a directed graph is *strongly connected* if, for every two nodes \( u \) and \( v \), there is a path from \( u \) to \( v \) and a path from \( v \) to \( u \).

In addition to simply knowing about the existence of a path between some pair of nodes \( u \) and \( v \), we may also want to know whether there is a *short* path. Thus we define the *distance* between two nodes \( u \) and \( v \) to be the minimum number of edges in a \( u\)-\( v \) path. (We can designate some symbol like \( \infty \) to denote the distance between nodes that are not connected by a path.) The term *distance* here comes from imagining \( G \) as representing a communication or transportation network; if we want to get from \( u \) to \( v \), we may well want a route with as few “hops” as possible.

**Trees** We say that an undirected graph is a *tree* if it is connected and does not contain a cycle. For example, the two graphs pictured in Figure 3.1 are trees. In a strong sense, trees are the simplest kind of connected graph: deleting any edge from a tree will disconnect it.

For thinking about the structure of a tree \( T \), it is useful to *root* it at a particular node \( r \). Physically, this is the operation of grabbing \( T \) at the node \( r \) and letting the rest of it hang downward under the force of gravity, like a mobile. More precisely, we “orient” each edge of \( T \) away from \( r \); for each other node \( v \), we declare the *parent* of \( v \) to be the node \( u \) that directly precedes \( v \) on its path from \( r \); we declare \( w \) to be a *child* of \( v \) if \( v \) is the parent of \( w \). More generally, we say that \( w \) is a *descendant* of \( v \) (or \( v \) is an *ancestor* of \( w \)) if \( v \) lies on the path from the root to \( w \); and we say that a node \( x \) is a *leaf* if it has no descendants. Thus, for example, the two pictures in Figure 3.1 correspond to the same tree \( T \)—the same pairs of nodes are joined by edges—but the drawing on the right represents the result of rooting \( T \) at node 1.
Rooted trees are fundamental objects in computer science, because they encode the notion of a hierarchy. For example, we can imagine the rooted tree in Figure 3.1 as corresponding to the organizational structure of a tiny nine-person company; employees 3 and 4 report to employee 2; employees 2, 5, and 7 report to employee 1; and so on. Many Web sites are organized according to a tree-like structure, to facilitate navigation. A typical computer science department’s Web site will have an entry page as the root; the People page is a child of this entry page (as is the Courses page); pages entitled Faculty and Students are children of the People page; individual professors’ home pages are children of the Faculty page; and so on.

For our purposes here, rooting a tree $T$ can make certain questions about $T$ conceptually easy to answer. For example, given a tree $T$ on $n$ nodes, how many edges does it have? Each node other than the root has a single edge leading “upward” to its parent; and conversely, each edge leads upward from precisely one non-root node. Thus we have very easily proved the following fact.

**(3.1)** Every $n$-node tree has exactly $n - 1$ edges.

In fact, the following stronger statement is true, although we do not prove it here.

**(3.2)** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements implies the third.

(i) $G$ is connected.
(ii) $G$ does not contain a cycle.
(iii) $G$ has $n - 1$ edges.

We now turn to the role of trees in the fundamental algorithmic idea of graph traversal.

### 3.2 Graph Connectivity and Graph Traversal

Having built up some fundamental notions regarding graphs, we turn to a very basic algorithmic question: node-to-node connectivity. Suppose we are given a graph $G = (V, E)$ and two particular nodes $s$ and $t$. We’d like to find an efficient algorithm that answers the question: Is there a path from $s$ to $t$ in $G$? We will call this the problem of determining $s$-$t$ connectivity.

For very small graphs, this question can often be answered easily by visual inspection. But for large graphs, it can take some work to search for a path. Indeed, the $s$-$t$ Connectivity Problem could also be called the Maze-Solving Problem. If we imagine $G$ as a maze with a room corresponding to each node, and a hallway corresponding to each edge that joins nodes (rooms) together,
then the problem is to start in a room \( s \) and find your way to another designated room \( t \). How efficient an algorithm can we design for this task?

In this section, we describe two natural algorithms for this problem at a high level: breadth-first search (BFS) and depth-first search (DFS). In the next section we discuss how to implement each of these efficiently, building on a data structure for representing a graph as the input to an algorithm.

**Breadth-First Search**

Perhaps the simplest algorithm for determining \( s-t \) connectivity is *breadth-first search* (BFS), in which we explore outward from \( s \) in all possible directions, adding nodes one “layer” at a time. Thus we start with \( s \) and include all nodes that are joined by an edge to \( s \)—this is the first layer of the search. We then include all additional nodes that are joined by an edge to any node in the first layer—this is the second layer. We continue in this way until no new nodes are encountered.

In the example of Figure 3.2, starting with node 1 as \( s \), the first layer of the search would consist of nodes 2 and 3, the second layer would consist of nodes 4, 5, 7, and 8, and the third layer would consist just of node 6. At this point the search would stop, since there are no further nodes that could be added (and in particular, note that nodes 9 through 13 are never reached by the search).

As this example reinforces, there is a natural physical interpretation to the algorithm. Essentially, we start at \( s \) and “flood” the graph with an expanding wave that grows to visit all nodes that it can reach. The layer containing a node represents the point in time at which the node is reached.

We can define the layers \( L_1, L_2, L_3, \ldots \) constructed by the BFS algorithm more precisely as follows.
Chapter 3  Graphs

- Layer $L_1$ consists of all nodes that are neighbors of $s$. (For notational reasons, we will sometimes use layer $L_0$ to denote the set consisting just of $s$.)
- Assuming that we have defined layers $L_1, \ldots, L_j$, then layer $L_{j+1}$ consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer $L_j$.

Recalling our definition of the distance between two nodes as the minimum number of edges on a path joining them, we see that layer $L_1$ is the set of all nodes at distance 1 from $s$, and more generally layer $L_j$ is the set of all nodes at distance exactly $j$ from $s$. A node fails to appear in any of the layers if and only if there is no path to it. Thus, BFS is not only determining the nodes that $s$ can reach, it is also computing shortest paths to them. We sum this up in the following fact.

\[
(3.3) \quad \text{For each } j \geq 1, \text{ layer } L_j \text{ produced by BFS consists of all nodes at distance exactly } j \text{ from } s. \text{ There is a path from } s \text{ to } t \text{ if and only if } t \text{ appears in some layer.}
\]

A further property of breadth-first search is that it produces, in a very natural way, a tree $T$ rooted at $s$ on the set of nodes reachable from $s$. Specifically, for each such node $v$ (other than $s$), consider the moment when $v$ is first “discovered” by the BFS algorithm; this happens when some node $u$ in layer $L_j$ is being examined, and we find that it has an edge to the previously unseen node $v$. At this moment, we add the edge $(u, v)$ to the tree $T$—$u$ becomes the parent of $v$, representing the fact that $u$ is “responsible” for completing the path to $v$. We call the tree $T$ that is produced in this way a breadth-first search tree.

Figure 3.3 depicts the construction of a BFS tree rooted at node 1 for the graph in Figure 3.2. The solid edges are the edges of $T$; the dotted edges are edges of $G$ that do not belong to $T$. The execution of BFS that produces this tree can be described as follows.

(a) Starting from node 1, layer $L_1$ consists of the nodes $\{2, 3\}$.
(b) Layer $L_2$ is then grown by considering the nodes in layer $L_1$ in order (say, first 2, then 3). Thus we discover nodes 4 and 5 as soon as we look at 2, so 2 becomes their parent. When we consider node 2, we also discover an edge to 3, but this isn’t added to the BFS tree, since we already know about node 3.

We first discover nodes 7 and 8 when we look at node 3. On the other hand, the edge from 3 to 5 is another edge of $G$ that does not end up in
3.2 Graph Connectivity and Graph Traversal

Figure 3.3 The construction of a breadth-first search tree $T$ for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of $T$; the dotted edges are in the connected component of $G$ containing node 1, but do not belong to $T$.

(a) (b) (c)

the BFS tree, because by the time we look at this edge out of node 3, we already know about node 5.

(c) We then consider the nodes in layer $L_2$ in order, but the only new node discovered when we look through $L_2$ is node 6, which is added to layer $L_3$. Note that the edges $(4, 5)$ and $(7, 8)$ don’t get added to the BFS tree, because they don’t result in the discovery of new nodes.

(d) No new nodes are discovered when node 6 is examined, so nothing is put in layer $L_4$, and the algorithm terminates. The full BFS tree is depicted in Figure 3.3(c).

We notice that as we ran BFS on this graph, the nontree edges all either connected nodes in the same layer, or connected nodes in adjacent layers. We now prove that this is a property of BFS trees in general.

(3.4) Let $T$ be a breadth-first search tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$ respectively, and let $(x, y)$ be an edge of $G$. Then $i$ and $j$ differ by at most 1.

Proof. Suppose by way of contradiction that $i$ and $j$ differed by more than 1; in particular, suppose $i < j - 1$. Now consider the point in the BFS algorithm when the edges incident to $x$ were being examined. Since $x$ belongs to layer $L_i$, the only nodes discovered from $x$ belong to layers $L_{i+1}$ and earlier; hence, if $y$ is a neighbor of $x$, then it should have been discovered by this point at the latest and hence should belong to layer $L_{i+1}$ or earlier. ■
Exploring a Connected Component

The set of nodes discovered by the BFS algorithm is precisely those reachable from the starting node $s$. We will refer to this set $R$ as the connected component of $G$ containing $s$; and once we know the connected component containing $s$, we can simply check whether $t$ belongs to it so as to answer the question of $s$-$t$ connectivity.

Now, if one thinks about it, it’s clear that BFS is just one possible way to produce this component. At a more general level, we can build the component $R$ by “exploring” $G$ in any order, starting from $s$. To start off, we define $R = \{s\}$. Then at any point in time, if we find an edge $(u, v)$ where $u \in R$ and $v \notin R$, we can add $v$ to $R$. Indeed, if there is a path $P$ from $s$ to $u$, then there is a path from $s$ to $v$ obtained by first following $P$ and then following the edge $(u, v)$. Figure 3.4 illustrates this basic step in growing the component $R$.

Suppose we continue growing the set $R$ until there are no more edges leading out of $R$; in other words, we run the following algorithm.

---

**R will consist of nodes to which $s$ has a path**

**Initially** $R = \{s\}$

**While there is an edge $(u, v)$ where $u \in R$ and $v \notin R**

  *Add $v$ to $R$*

**Endwhile**

---

Here is the key property of this algorithm.

**(3.5)** *The set $R$ produced at the end of the algorithm is precisely the connected component of $G$ containing $s$.***
3.2 Graph Connectivity and Graph Traversal

Proof. We have already argued that for any node $v \in R$, there is a path from $s$ to $v$.

Now, consider a node $w \notin R$, and suppose by way of contradiction, that there is an $s$-$w$ path $P$ in $G$. Since $s \in R$ but $w \notin R$, there must be a first node $v$ on $P$ that does not belong to $R$; and this node $v$ is not equal to $s$. Thus there is a node $u$ immediately preceding $v$ on $P$, so $(u, v)$ is an edge. Moreover, since $v$ is the first node on $P$ that does not belong to $R$, we must have $u \in R$. It follows that $(u, v)$ is an edge where $u \in R$ and $v \notin R$; this contradicts the stopping rule for the algorithm.

For any node $t$ in the component $R$, observe that it is easy to recover the actual path from $s$ to $t$ along the lines of the argument above: we simply record, for each node $v$, the edge $(u, v)$ that was considered in the iteration in which $v$ was added to $R$. Then, by tracing these edges backward from $t$, we proceed through a sequence of nodes that were added in earlier and earlier iterations, eventually reaching $s$; this defines an $s$-$t$ path.

To conclude, we notice that the general algorithm we have defined to grow $R$ is underspecified, so how do we decide which edge to consider next? The BFS algorithm arises, in particular, as a particular way of ordering the nodes we visit—in successive layers, based on their distance from $s$. But there are other natural ways to grow the component, several of which lead to efficient algorithms for the connectivity problem while producing search patterns with different structures. We now go on to discuss a different one of these algorithms, depth-first search, and develop some of its basic properties.

Depth-First Search

Another natural method to find the nodes reachable from $s$ is the approach you might take if the graph $G$ were truly a maze of interconnected rooms and you were walking around in it. You’d start from $s$ and try the first edge leading out of it, to a node $v$. You’d then follow the first edge leading out of $v$, and continue in this way until you reached a “dead end”—a node for which you had already explored all its neighbors. You’d then backtrack until you got to a node with an unexplored neighbor, and resume from there. We call this algorithm depth-first search (DFS), since it explores $G$ by going as deeply as possible and only retreating when necessary.

DFS is also a particular implementation of the generic component-growing algorithm that we introduced earlier. It is most easily described in recursive form: we can invoke DFS from any starting point but maintain global knowledge of which nodes have already been explored.

For any query, contact us at help@tutorialsduniya.com
Chapter 3  Graphs

DFS(u):
Mark u as "Explored" and add u to R
For each edge (u, v) incident to u
  If v is not marked "Explored" then
    Recursively invoke DFS(v)
  Endif
Endfor

To apply this to s-t connectivity, we simply declare all nodes initially to be not explored, and invoke DFS(s).

There are some fundamental similarities and some fundamental differences between DFS and BFS. The similarities are based on the fact that they both build the connected component containing s, and we will see in the next section that they achieve qualitatively similar levels of efficiency.

While DFS ultimately visits exactly the same set of nodes as BFS, it typically does so in a very different order; it probes its way down long paths, potentially getting very far from s, before backing up to try nearer unexplored nodes. We can see a reflection of this difference in the fact that, like BFS, the DFS algorithm yields a natural rooted tree T on the component containing s, but the tree will generally have a very different structure. We make s the root of the tree T, and make u the parent of v when u is responsible for the discovery of v. That is, whenever DFS(v) is invoked directly during the call to DFS(u), we add the edge (u, v) to T. The resulting tree is called a depth-first search tree of the component R.

Figure 3.5 depicts the construction of a DFS tree rooted at node 1 for the graph in Figure 3.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T. The execution of DFS begins by building a path on nodes 1, 2, 3, 5, 4. The execution reaches a dead end at 4, since there are no new nodes to find, and so it “backs up” to 5, finds node 6, backs up again to 3, and finds nodes 7 and 8. At this point there are no new nodes to find in the connected component, so all the pending recursive DFS calls terminate, one by one, and the execution comes to an end. The full DFS tree is depicted in Figure 3.5(g).

This example suggests the characteristic way in which DFS trees look different from BFS trees. Rather than having root-to-leaf paths that are as short as possible, they tend to be quite narrow and deep. However, as in the case of BFS, we can say something quite strong about the way in which nontree edges of G must be arranged relative to the edges of a DFS tree T: as in the figure, nontree edges can only connect ancestors of T to descendants.
To establish this, we first observe the following property of the DFS algorithm and the tree that it produces.

**3.6** For a given recursive call DFS(u), all nodes that are marked “Explored” between the invocation and end of this recursive call are descendants of u in T.

Using (3.6), we prove

**3.7** Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.
Proof. Suppose that \((x, y)\) is an edge of \(G\) that is not an edge of \(T\), and suppose without loss of generality that \(x\) is reached first by the DFS algorithm. When the edge \((x, y)\) is examined during the execution of \(DFS(x)\), it is not added to \(T\) because \(y\) is marked “Explored.” Since \(y\) was not marked “Explored” when \(DFS(x)\) was first invoked, it is a node that was discovered between the invocation and end of the recursive call \(DFS(x)\). It follows from (3.6) that \(y\) is a descendant of \(x\).  

The Set of All Connected Components

So far we have been talking about the connected component containing a particular node \(s\). But there is a connected component associated with each node in the graph. What is the relationship between these components?

In fact, this relationship is highly structured and is expressed in the following claim.

\[(3.8) \quad \text{For any two nodes } s \text{ and } t \text{ in a graph, their connected components are either identical or disjoint.}\]

This is a statement that is very clear intuitively, if one looks at a graph like the example in Figure 3.2. The graph is divided into multiple pieces with no edges between them; the largest piece is the connected component of nodes 1 through 8, the medium piece is the connected component of nodes 11, 12, and 13, and the smallest piece is the connected component of nodes 9 and 10. To prove the statement in general, we just need to show how to define these “pieces” precisely for an arbitrary graph.

Proof. Consider any two nodes \(s\) and \(t\) in a graph \(G\) with the property that there is a path between \(s\) and \(t\). We claim that the connected components containing \(s\) and \(t\) are the same set. Indeed, for any node \(v\) in the component of \(s\), the node \(v\) must also be reachable from \(t\) by a path: we can just walk from \(t\) to \(s\), and then on from \(s\) to \(v\). The same reasoning works with the roles of \(s\) and \(t\) reversed, and so a node is in the component of one if and only if it is in the component of the other.

On the other hand, if there is no path between \(s\) and \(t\), then there cannot be a node \(v\) that is in the connected component of each. For if there were such a node \(v\), then we could walk from \(s\) to \(v\) and then on to \(t\), constructing a path between \(s\) and \(t\). Thus, if there is no path between \(s\) and \(t\), then their connected components are disjoint.  

This proof suggests a natural algorithm for producing all the connected components of a graph, by growing them one component at a time. We start with an arbitrary node \(s\), and we use BFS (or DFS) to generate its connected
3.3 Implementing Graph Traversal Using Queues and Stacks

We then find a node \( v \) (if any) that was not visited by the search from \( s \), and iterate, using BFS starting from \( v \), to generate its connected component—which, by (3.8), will be disjoint from the component of \( s \). We continue in this way until all nodes have been visited.

3.3 Implementing Graph Traversal Using Queues and Stacks

So far we have been discussing basic algorithmic primitives for working with graphs without mentioning any implementation details. Here we discuss how to use lists and arrays to represent graphs, and we discuss the trade-offs between the different representations. Then we use these data structures to implement the graph traversal algorithms breadth-first search (BFS) and depth-first search (DFS) efficiently. We will see that BFS and DFS differ essentially only in that one uses a queue and the other uses a stack, two simple data structures that we will describe later in this section.

Representing Graphs

There are two basic ways to represent graphs: by an adjacency matrix and by an adjacency list representation. Throughout the book we will use the adjacency list representation. We start, however, by reviewing both of these representations and discussing the trade-offs between them.

A graph \( G = (V, E) \) has two natural input parameters, the number of nodes \( |V| \), and the number of edges \( |E| \). We will use \( n = |V| \) and \( m = |E| \) to denote these, respectively. Running times will be given in terms of both of these two parameters. As usual, we will aim for polynomial running times, and lower-degreet polynomials are better. However, with two parameters in the running time, the comparison is not always so clear. Is \( O(m^2) \) or \( O(n^3) \) a better running time? This depends on what the relation is between \( n \) and \( m \). With at most one edge between any pair of nodes, the number of edges \( m \) can be at most \( \binom{n}{2} \leq n^2 \). On the other hand, in many applications the graphs of interest are connected, and by (3.1), connected graphs must have at least \( m \geq n - 1 \) edges. But these comparisons do not always tell us which of two running times (such as \( m^2 \) and \( n^3 \)) are better, so we will tend to keep the running times in terms of both of these parameters. In this section we aim to implement the basic graph search algorithms in time \( O(m + n) \). We will refer to this as linear time, since it takes \( O(m + n) \) time simply to read the input. Note that when we work with connected graphs, a running time of \( O(m + n) \) is the same as \( O(m) \), since \( m \geq n - 1 \).

Consider a graph \( G = (V, E) \) with \( n \) nodes, and assume the set of nodes is \( V = \{1, \ldots, n\} \). The simplest way to represent a graph is by an adjacency
matrix, which is an $n \times n$ matrix $A$ where $A[u, v]$ is equal to 1 if the graph contains the edge $(u, v)$ and 0 otherwise. If the graph is undirected, the matrix $A$ is symmetric, with $A[u, v] = A[v, u]$ for all nodes $u, v \in V$. The adjacency matrix representation allows us to check in $O(1)$ time if a given edge $(u, v)$ is present in the graph. However, the representation has two basic disadvantages.

- The representation takes $\Theta(n^2)$ space. When the graph has many fewer edges than $n^2$, more compact representations are possible.
- Many graph algorithms need to examine all edges incident to a given node $v$. In the adjacency matrix representation, doing this involves considering all other nodes $w$, and checking the matrix entry $A[v, w]$ to see whether the edge $(v, w)$ is present—and this takes $\Theta(n)$ time. In the worst case, $v$ may have $\Theta(n)$ incident edges, in which case checking all these edges will take $\Theta(n)$ time regardless of the representation. But many graphs in practice have significantly fewer edges incident to most nodes, and so it would be good to be able to find all these incident edges more efficiently.

The representation of graphs used throughout the book is the adjacency list, which works better for sparse graphs—that is, those with many fewer than $n^2$ edges. In the adjacency list representation there is a record for each node $v$, containing a list of the nodes to which $v$ has edges. To be precise, we have an array Adj, where Adj[$v$] is a record containing a list of all nodes adjacent to node $v$. For an undirected graph $G = (V, E)$, each edge $e = (v, w) \in E$ occurs on two adjacency lists: node $w$ appears on the list for node $v$, and node $v$ appears on the list for node $w$.

Let’s compare the adjacency matrix and adjacency list representations. First consider the space required by the representation. An adjacency matrix requires $O(n^2)$ space, since it uses an $n \times n$ matrix. In contrast, we claim that the adjacency list representation requires only $O(m + n)$ space. Here is why. First, we need an array of pointers of length $n$ to set up the lists in Adj, and then we need space for all the lists. Now, the lengths of these lists may differ from node to node, but we argued in the previous paragraph that overall, each edge $e = (v, w)$ appears in exactly two of the lists: the one for $v$ and the one for $w$. Thus the total length of all lists is $2m = O(m)$.

Another (essentially equivalent) way to justify this bound is as follows. We define the degree $n_v$ of a node $v$ to be the number of incident edges it has. The length of the list at Adj[$v$] is list is $n_v$, so the total length over all nodes is $O(\sum_{v \in V} n_v)$. Now, the sum of the degrees in a graph is a quantity that often comes up in the analysis of graph algorithms, so it is useful to work out what this sum is.

$$\sum_{v \in V} n_v = 2m.$$
3.3 Implementing Graph Traversal Using Queues and Stacks

**Proof.** Each edge \( e = (v, w) \) contributes exactly twice to this sum: once in the quantity \( n_v \) and once in the quantity \( n_w \). Since the sum is the total of the contributions of each edge, it is \( 2m \).

We sum up the comparison between adjacency matrices and adjacency lists as follows.

\[ (3.10) \quad \text{The adjacency matrix representation of a graph requires } O(n^2) \text{ space, while the adjacency list representation requires only } O(m + n) \text{ space.} \]

Since we have already argued that \( m \leq n^2 \), the bound \( O(m + n) \) is never worse than \( O(n^2) \); and it is much better when the underlying graph is sparse, with \( m \) much smaller than \( n^2 \).

Now we consider the ease of accessing the information stored in these two different representations. Recall that in an adjacency matrix we can check in \( O(1) \) time if a particular edge \( (u, v) \) is present in the graph. In the adjacency list representation, this can take time proportional to the degree \( O(n_v) \): we have to follow the pointers on \( u \)'s adjacency list to see if edge \( v \) occurs on the list. On the other hand, if the algorithm is currently looking at a node \( u \), it can read the list of neighbors in constant time per neighbor.

In view of this, the adjacency list is a natural representation for exploring graphs. If the algorithm is currently looking at a node \( u \), it can read this list of neighbors in constant time per neighbor; move to a neighbor \( v \) once it encounters it on this list in constant time; and then be ready to read the list associated with node \( v \). The list representation thus corresponds to a physical notion of “exploring” the graph, in which you learn the neighbors of a node \( u \) once you arrive at \( u \), and can read them off in constant time per neighbor.

**Queues and Stacks**

Many algorithms have an inner step in which they need to process a set of elements, such as the set of all edges adjacent to a node in a graph, the set of visited nodes in BFS and DFS, or the set of all free men in the Stable Matching algorithm. For this purpose, it is natural to maintain the set of elements to be considered in a linked list, as we have done for maintaining the set of free men in the Stable Matching algorithm.

One important issue that arises is the order in which to consider the elements in such a list. In the Stable Matching algorithm, the order in which we considered the free men did not affect the outcome, although this required a fairly subtle proof to verify. In many other algorithms, such as DFS and BFS, the order in which elements are considered is crucial.
Two of the simplest and most natural options are to maintain a set of elements as either a queue or a stack. A *queue* is a set from which we extract elements in *first-in, first-out* (FIFO) order: we select elements in the same order in which they were added. A *stack* is a set from which we extract elements in *last-in, first-out* (LIFO) order: each time we select an element, we choose the one that was added most recently. Both queues and stacks can be easily implemented via a doubly linked list. In both cases, we always select the first element on our list; the difference is in where we insert a new element. In a queue a new element is added to the end of the list as the last element, while in a stack a new element is placed in the first position on the list. Recall that a doubly linked list has explicit First and Last pointers to the beginning and end, respectively, so each of these insertions can be done in constant time.

Next we will discuss how to implement the search algorithms of the previous section in linear time. We will see that BFS can be thought of as using a queue to select which node to consider next, while DFS is effectively using a stack.

### Implementing Breadth-First Search

The adjacency list data structure is ideal for implementing breadth-first search. The algorithm examines the edges leaving a given node one by one. When we are scanning the edges leaving \( u \) and come to an edge \((u, v)\), we need to know whether or not node \( v \) has been previously discovered by the search. To make this simple, we maintain an array \( \text{Discovered} \) of length \( n \) and set \( \text{Discovered}[v] = \text{true} \) as soon as our search first sees \( v \). The algorithm, as described in the previous section, constructs layers of nodes \( L_1, L_2, \ldots \), where \( L_i \) is the set of nodes at distance \( i \) from the source \( s \). To maintain the nodes in a layer \( L_i \), we have a list \( L[i] \) for each \( i = 0, 1, 2, \ldots \).

---

**BFS\((s)\):**

Set \( \text{Discovered}[s] = \text{true} \) and \( \text{Discovered}[v] = \text{false} \) for all other \( v \)

Initialize \( L[0] \) to consist of the single element \( s \)

Set the layer counter \( i = 0 \)

Set the current BFS tree \( T = \emptyset \)

While \( L[i] \) is not empty

- Initialize an empty list \( L[i+1] \)
- For each node \( u \in L[i] \)
  - Consider each edge \((u, v)\) incident to \( u \)
  - If \( \text{Discovered}[v] = \text{false} \) then
    - Set \( \text{Discovered}[v] = \text{true} \)
    - Add edge \((u, v)\) to the tree \( T \)
3.3 Implementing Graph Traversal Using Queues and Stacks

Add \( v \) to the list \( L[i + 1] \)
Endif
Endfor
Increment the layer counter \( i \) by one
Endwhile

In this implementation it does not matter whether we manage each list \( L[i] \) as a queue or a stack, since the algorithm is allowed to consider the nodes in a layer \( L_i \) in any order.

**3.11** The above implementation of the BFS algorithm runs in time \( O(m + n) \) (i.e., linear in the input size), if the graph is given by the adjacency list representation.

**Proof.** As a first step, it is easy to bound the running time of the algorithm by \( O(n^2) \) (a weaker bound than our claimed \( O(m + n) \)). To see this, note that there are at most \( n \) lists \( L[i] \) that we need to set up, so this takes \( O(n) \) time. Now we need to consider the nodes \( u \) on these lists. Each node occurs on at most one list, so the For loop runs at most \( n \) times over all iterations of the While loop. When we consider a node \( u \), we need to look through all edges \((u, v)\) incident to \( u \). There can be at most \( n \) such edges, and we spend \( O(1) \) time considering each edge. So the total time spent on one iteration of the For loop is at most \( O(n) \). We’ve thus concluded that there are at most \( n \) iterations of the For loop, and that each iteration takes at most \( O(n) \) time, so the total time is at most \( O(n^2) \).

To get the improved \( O(m + n) \) time bound, we need to observe that the For loop processing a node \( u \) can take less than \( O(n) \) time if \( u \) has only a few neighbors. As before, let \( n_u \) denote the degree of node \( u \), the number of edges incident to \( u \). Now, the time spent in the For loop considering edges incident to node \( u \) is \( O(n_u) \), so the total over all nodes is \( O(\sum_{u \in V} n_u) \). Recall from (3.9) that \( \sum_{u \in V} n_u = 2m \), and so the total time spent considering edges over the whole algorithm is \( O(m) \). We need \( O(n) \) additional time to set up lists and manage the array Discovered. So the total time spent is \( O(m + n) \) as claimed.

We described the algorithm using up to \( n \) separate lists \( L[i] \) for each layer \( L_i \). Instead of all these distinct lists, we can implement the algorithm using a single list \( L \) that we maintain as a queue. In this way, the algorithm processes nodes in the order they are first discovered: each time a node is discovered, it is added to the end of the queue, and the algorithm always processes the edges out of the node that is currently first in the queue.
Chapter 3  Graphs

If we maintain the discovered nodes in this order, then all nodes in layer $L_i$ will appear in the queue ahead of all nodes in layer $L_{i+1}$, for $i = 0, 1, 2 \ldots$. Thus, all nodes in layer $L_i$ will be considered in a contiguous sequence, followed by all nodes in layer $L_{i+1}$, and so forth. Hence this implementation in terms of a single queue will produce the same result as the BFS implementation above.

Implementing Depth-First Search

We now consider the depth-first search algorithm. In the previous section we presented DFS as a recursive procedure, which is a natural way to specify it. However, it can also be viewed as almost identical to BFS, with the difference that it maintains the nodes to be processed in a stack, rather than in a queue. Essentially, the recursive structure of DFS can be viewed as pushing nodes onto a stack for later processing, while moving on to more freshly discovered nodes. We now show how to implement DFS by maintaining this stack of nodes to be processed explicitly.

In both BFS and DFS, there is a distinction between the act of discovering a node $v$—the first time it is seen, when the algorithm finds an edge leading to $v$—and the act of exploring a node $v$, when all the incident edges to $v$ are scanned, resulting in the potential discovery of further nodes. The difference between BFS and DFS lies in the way in which discovery and exploration are interleaved.

In BFS, once we started to explore a node $u$ in layer $L_i$, we added all its newly discovered neighbors to the next layer $L_{i+1}$, and we deferred actually exploring these neighbors until we got to the processing of layer $L_{i+1}$. In contrast, DFS is more impulsive: when it explores a node $u$, it scans the neighbors of $u$ until it finds the first not-yet-explored node $v$ (if any), and then it immediately shifts attention to exploring $v$.

To implement the exploration strategy of DFS, we first add all of the nodes adjacent to $u$ to our list of nodes to be considered, but after doing this we proceed to explore a new neighbor $v$ of $u$. As we explore $v$, in turn, we add the neighbors of $v$ to the list we’re maintaining, but we do so in stack order, so that these neighbors will be explored before we return to explore the other neighbors of $u$. We only come back to other nodes adjacent to $u$ when there are no other nodes left.

In addition, we use an array $\text{Explored}$ analogous to the $\text{Discovered}$ array we used for BFS. The difference is that we only set $\text{Explored}[v]$ to be $\text{true}$ when we scan $v$’s incident edges (when the DFS search is at $v$), while BFS sets $\text{Discovered}[v]$ to $\text{true}$ as soon as $v$ is first discovered. The implementation in full looks as follows.
DFS(s):
Initialize $S$ to be a stack with one element $s$
While $S$ is not empty
  Take a node $u$ from $S$
  If Explored[$u$] = false then
    Set Explored[$u$] = true
    For each edge $(u,v)$ incident to $u$
      Add $v$ to the stack $S$
  Endfor
  Endif
Endwhile

There is one final wrinkle to mention. Depth-first search is underspecified, since the adjacency list of a node being explored can be processed in any order. Note that the above algorithm, because it pushes all adjacent nodes onto the stack before considering any of them, in fact processes each adjacency list in the reverse order relative to the recursive version of DFS in the previous section.

(3.12) The above algorithm implements DFS, in the sense that it visits the nodes in exactly the same order as the recursive DFS procedure in the previous section (except that each adjacency list is processed in reverse order).

If we want the algorithm to also find the DFS tree, we need to have each node $u$ on the stack $S$ maintain the node that “caused” $u$ to get added to the stack. This can be easily done by using an array parent and setting parent[$v$] = $u$ when we add node $v$ to the stack due to edge $(u,v)$. When we mark a node $u \neq s$ as Explored, we also can add the edge $(u,\text{parent}[u])$ to the tree $T$. Note that a node $v$ may be in the stack $S$ multiple times, as it can be adjacent to multiple nodes $u$ that we explore, and each such node adds a copy of $v$ to the stack $S$. However, we will only use one of these copies to explore node $v$, the copy that we add last. As a result, it suffices to maintain one value parent[$v$] for each node $v$ by simply overwriting the value parent[$v$] every time we add a new copy of $v$ to the stack $S$.

The main step in the algorithm is to add and delete nodes to and from the stack $S$, which takes $O(1)$ time. Thus, to bound the running time, we need to bound the number of these operations. To count the number of stack operations, it suffices to count the number of nodes added to $S$, as each node needs to be added once for every time it can be deleted from $S$.

How many elements ever get added to $S$? As before, let $n_v$ denote the degree of node $v$. Node $v$ will be added to the stack $S$ every time one of its $n_v$ adjacent nodes is explored, so the total number of nodes added to $S$ is at
most $\sum_u n_u = 2m$. This proves the desired $O(m + n)$ bound on the running time of DFS.

(3.13) *The above implementation of the DFS algorithm runs in time $O(m + n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.*

**Finding the Set of All Connected Components**

In the previous section we talked about how one can use BFS (or DFS) to find all connected components of a graph. We start with an arbitrary node $s$, and we use BFS (or DFS) to generate its connected component. We then find a node $v$ (if any) that was not visited by the search from $s$ and iterate, using BFS (or DFS) starting from $v$ to generate its connected component—which, by (3.8), will be disjoint from the component of $s$. We continue in this way until all nodes have been visited.

Although we earlier expressed the running time of BFS and DFS as $O(m + n)$, where $m$ and $n$ are the total number of edges and nodes in the graph, both BFS and DFS in fact spend work only on edges and nodes in the connected component containing the starting node. (They never see any of the other nodes or edges.) Thus the above algorithm, although it may run BFS or DFS a number of times, only spends a constant amount of work on a given edge or node in the iteration when the connected component it belongs to is under consideration. Hence the overall running time of this algorithm is still $O(m + n)$.

**3.4 Testing Bipartiteness: An Application of Breadth-First Search**

Recall the definition of a bipartite graph: it is one where the node set $V$ can be partitioned into sets $X$ and $Y$ in such a way that every edge has one end in $X$ and the other end in $Y$. To make the discussion a little smoother, we can imagine that the nodes in the set $X$ are colored red, and the nodes in the set $Y$ are colored blue. With this imagery, we can say a graph is bipartite if it is possible to color its nodes red and blue so that every edge has one red end and one blue end.

**The Problem**

In the earlier chapters, we saw examples of bipartite graphs. Here we start by asking: What are some natural examples of a nonbipartite graph, one where no such partition of $V$ is possible?
3.4 Testing Bipartiteness: An Application of Breadth-First Search

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can’t do anything with the third node. More generally, consider a cycle $C$ of odd length, with nodes numbered $1, 2, 3, \ldots, 2k, 2k + 1$. If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on—coloring odd-numbered nodes red and even-numbered nodes blue. But then we must color node $2k + 1$ red, and it has an edge to node 1, which is also red. This demonstrates that there’s no way to partition $C$ into red and blue nodes as required. More generally, if a graph $G$ simply contains an odd cycle, then we can apply the same argument; thus we have established the following.

**3.14** If a graph $G$ is bipartite, then it cannot contain an odd cycle.

It is easy to recognize that a graph is bipartite when appropriate sets $X$ and $Y$ (i.e., red and blue nodes) have actually been identified for us; and in many settings where bipartite graphs arise, this is natural. But suppose we encounter a graph $G$ with no annotation provided for us, and we’d like to determine for ourselves whether it is bipartite—that is, whether there exists a partition into red and blue nodes, as required. How difficult is this? We see from (3.14) that an odd cycle is one simple “obstacle” to a graph’s being bipartite. Are there other, more complex obstacles to bipartiteness?

**Designing the Algorithm**

In fact, there is a very simple procedure to test for bipartiteness, and its analysis can be used to show that odd cycles are the only obstacle. First we assume the graph $G$ is connected, since otherwise we can first compute its connected components and analyze each of them separately. Next we pick any node $s \in V$ and color it red; there is no loss in doing this, since $s$ must receive some color. It follows that all the neighbors of $s$ must be colored blue, so we do this. It then follows that all the neighbors of these nodes must be colored red, their neighbors must be colored blue, and so on, until the whole graph is colored. At this point, either we have a valid red/blue coloring of $G$, in which every edge has ends of opposite colors, or there is some edge with ends of the same color. In this latter case, it seems clear that there’s nothing we could have done: $G$ simply is not bipartite. We now want to argue this point precisely and also work out an efficient way to perform the coloring.

The first thing to notice is that the coloring procedure we have just described is essentially identical to the description of BFS: we move outward from $s$, coloring nodes as soon as we first encounter them. Indeed, another way to describe the coloring algorithm is as follows: we perform BFS, coloring
The cycle through \( x, y, \) and \( z \) has odd length.

We can implement this on top of BFS, by simply taking the implementation of BFS and adding an extra array \( \text{Color} \) over the nodes. Whenever we get to a step in BFS where we are adding a node \( v \) to a list \( L[i + 1] \), we assign \( \text{Color}[v] = \text{red} \) if \( i + 1 \) is an even number, and \( \text{Color}[v] = \text{blue} \) if \( i + 1 \) is an odd number. At the end of this procedure, we simply scan all the edges and determine whether there is any edge for which both ends received the same color. Thus, the total running time for the coloring algorithm is \( O(m + n) \), just as it is for BFS.

**Analyzing the Algorithm**

We now prove a claim that shows this algorithm correctly determines whether \( G \) is bipartite, and it also shows that we can find an odd cycle in \( G \) whenever it is not bipartite.

\[
(3.15) \quad \text{Let } G \text{ be a connected graph, and let } L_1, L_2, \ldots \text{ be the layers produced by BFS starting at node } s. \text{ Then exactly one of the following two things must hold.}

(i) There is no edge of } G \text{ joining two nodes of the same layer. In this case } G \text{ is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.}

(ii) There is an edge of } G \text{ joining two nodes of the same layer. In this case, } G \text{ contains an odd-length cycle, and so it cannot be bipartite.}
\]

**Proof.** First consider case (i), where we suppose that there is no edge joining two nodes of the same layer. By (3.4), we know that every edge of \( G \) joins nodes either in the same layer or in adjacent layers. Our assumption for case (i) is precisely that the first of these two alternatives never happens, so this means that every edge joins two nodes in adjacent layers. But our coloring procedure gives nodes in adjacent layers the opposite colors, and so every edge has ends with opposite colors. Thus this coloring establishes that \( G \) is bipartite.

Now suppose we are in case (ii); why must \( G \) contain an odd cycle? We are told that \( G \) contains an edge joining two nodes of the same layer. Suppose this is the edge \( e = (x, y) \), with \( x, y \in L_j \). Also, for notational reasons, recall that \( L_0 \) ("layer 0") is the set consisting of just \( s \). Now consider the BFS tree \( T \) produced by our algorithm, and let \( z \) be the node whose layer number is as large as possible, subject to the condition that \( z \) is an ancestor of both \( x \) and \( y \) in \( T \); for obvious reasons, we can call \( z \) the lowest common ancestor of \( x \) and \( y \). Suppose \( z \in L_i \), where \( i < j \). We now have the situation pictured in Figure 3.6. We consider the cycle \( C \) defined by following the \( z-x \) path in \( T \), then the edge \( e \),
3.5 Connectivity in Directed Graphs

Thus far, we have been looking at problems on undirected graphs; we now consider the extent to which these ideas carry over to the case of directed graphs.

Recall that in a directed graph, the edge \((u, v)\) has a direction: it goes from \(u\) to \(v\). In this way, the relationship between \(u\) and \(v\) is asymmetric, and this has qualitative effects on the structure of the resulting graph. In Section 3.1, for example, we discussed the World Wide Web as an instance of a large, complex directed graph whose nodes are pages and whose edges are hyperlinks. The act of browsing the Web is based on following a sequence of edges in this directed graph; and the directionality is crucial, since it’s not generally possible to browse “backwards” by following hyperlinks in the reverse direction.

At the same time, a number of basic definitions and algorithms have natural analogues in the directed case. This includes the adjacency list representation and graph search algorithms such as BFS and DFS. We now discuss these in turn.

Representing Directed Graphs

In order to represent a directed graph for purposes of designing algorithms, we use a version of the adjacency list representation that we employed for undirected graphs. Now, instead of each node having a single list of neighbors, each node has two lists associated with it: one list consists of nodes to which it has edges, and a second list consists of nodes from which it has edges. Thus an algorithm that is currently looking at a node \(u\) can read off the nodes reachable by going one step forward on a directed edge, as well as the nodes that would be reachable if one went one step in the reverse direction on an edge from \(u\).

The Graph Search Algorithms

Breadth-first search and depth-first search are almost the same in directed graphs as they are in undirected graphs. We will focus here on BFS. We start at a node \(s\), define a first layer of nodes to consist of all those to which \(s\) has an edge, define a second layer to consist of all additional nodes to which these first-layer nodes have an edge, and so forth. In this way, we discover nodes layer by layer as they are reached in this outward search from \(s\), and the nodes in layer \(j\) are precisely those for which the shortest path from \(s\) has exactly \(j\) edges. As in the undirected case, this algorithm performs at most constant work for each node and edge, resulting in a running time of \(O(m + n)\).
Chapter 3  Graphs

It is important to understand what this directed version of BFS is computing. In directed graphs, it is possible for a node \( s \) to have a path to a node \( t \) even though \( t \) has no path to \( s \); and what directed BFS is computing is the set of all nodes \( t \) with the property that \( s \) has a path to \( t \). Such nodes may or may not have paths back to \( s \).

There is a natural analogue of depth-first search as well, which also runs in linear time and computes the same set of nodes. It is again a recursive procedure that tries to explore as deeply as possible, in this case only following edges according to their inherent direction. Thus, when DFS is at a node \( u \), it recursively launches a depth-first search, in order, for each node to which \( u \) has an edge.

Suppose that, for a given node \( s \), we wanted the set of nodes with paths to \( s \), rather than the set of nodes to which \( s \) has paths. An easy way to do this would be to define a new directed graph, \( G^{\text{rev}} \), that we obtain from \( G \) simply by reversing the direction of every edge. We could then run BFS or DFS in \( G^{\text{rev}} \); a node has a path from \( s \) in \( G^{\text{rev}} \) if and only if it has a path to \( s \) in \( G \).

**Strong Connectivity**

Recall that a directed graph is strongly connected if, for every two nodes \( u \) and \( v \), there is a path from \( u \) to \( v \) and a path from \( v \) to \( u \). It’s worth also formulating some terminology for the property at the heart of this definition; let’s say that two nodes \( u \) and \( v \) in a directed graph are mutually reachable if there is a path from \( u \) to \( v \) and also a path from \( v \) to \( u \). (So a graph is strongly connected if every pair of nodes is mutually reachable.)

Mutual reachability has a number of nice properties, many of them stemming from the following simple fact.

(3.16)  If \( u \) and \( v \) are mutually reachable, and \( v \) and \( w \) are mutually reachable, then \( u \) and \( w \) are mutually reachable.

**Proof.** To construct a path from \( u \) to \( w \), we first go from \( u \) to \( v \) (along the path guaranteed by the mutual reachability of \( u \) and \( v \) ), and then on from \( v \) to \( w \) (along the path guaranteed by the mutual reachability of \( v \) and \( w \) ). To construct a path from \( w \) to \( u \), we just reverse this reasoning: we first go from \( w \) to \( v \) (along the path guaranteed by the mutual reachability of \( v \) and \( w \) ), and then on from \( v \) to \( u \) (along the path guaranteed by the mutual reachability of \( u \) and \( v \) ).

There is a simple linear-time algorithm to test if a directed graph is strongly connected, implicitly based on (3.16). We pick any node \( s \) and run BFS in \( G \) starting from \( s \). We then also run BFS starting from \( s \) in \( G^{\text{rev}} \). Now, if one of these two searches fails to reach every node, then clearly \( G \) is not strongly connected. But suppose we find that \( s \) has a path to every node, and that
every node has a path to $s$. Then $s$ and $v$ are mutually reachable for every $v$, and so it follows that every two nodes $u$ and $v$ are mutually reachable: $s$ and $u$ are mutually reachable, and $s$ and $v$ are mutually reachable, so by (3.16) we also have that $u$ and $v$ are mutually reachable.

By analogy with connected components in an undirected graph, we can define the strong component containing a node $s$ in a directed graph to be the set of all $v$ such that $s$ and $v$ are mutually reachable. If one thinks about it, the algorithm in the previous paragraph is really computing the strong component containing $s$: we run BFS starting from $s$ both in $G$ and in $G^{rev}$; the set of nodes reached by both searches is the set of nodes with paths to and from $s$, and hence this set is the strong component containing $s$.

There are further similarities between the notion of connected components in undirected graphs and strong components in directed graphs. Recall that connected components naturally partitioned the graph, since any two were either identical or disjoint. Strong components have this property as well, and for essentially the same reason, based on (3.16).

\[
\textbf{(3.17)} \quad \text{For any two nodes } s \text{ and } t \text{ in a directed graph, their strong components are either identical or disjoint.}
\]

**Proof.** Consider any two nodes $s$ and $t$ that are mutually reachable; we claim that the strong components containing $s$ and $t$ are identical. Indeed, for any node $v$, if $s$ and $v$ are mutually reachable, then by (3.16), $t$ and $v$ are mutually reachable as well. Similarly, if $t$ and $v$ are mutually reachable, then again by (3.16), $s$ and $v$ are mutually reachable.

On the other hand, if $s$ and $t$ are not mutually reachable, then there cannot be a node $v$ that is in the strong component of each. For if there were such a node $v$, then $s$ and $v$ would be mutually reachable, and $v$ and $t$ would be mutually reachable, so from (3.16) it would follow that $s$ and $t$ were mutually reachable.

In fact, although we will not discuss the details of this here, with more work it is possible to compute the strong components for all nodes in a total time of $O(m + n)$.

### 3.6 Directed Acyclic Graphs and Topological Ordering

If an undirected graph has no cycles, then it has an extremely simple structure: each of its connected components is a tree. But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure. For example, such graphs can have a large number of edges: if we start with the node
In a topological ordering, all edges point from left to right.

Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

set \( \{1, 2, \ldots, n\} \) and include an edge \((i, j)\) whenever \(i < j\), then the resulting directed graph has \( \binom{n}{2} \) edges but no cycles.

If a directed graph has no cycles, we call it—naturally enough—a directed acyclic graph, or a DAG for short. (The term DAG is typically pronounced as a word, not spelled out as an acronym.) In Figure 3.7(a) we see an example of a DAG, although it may take some checking to convince oneself that it really has no directed cycles.

The Problem

DAGs are a very common structure in computer science, because many kinds of dependency networks of the type we discussed in Section 3.1 are acyclic. Thus DAGs can be used to encode precedence relations or dependencies in a natural way. Suppose we have a set of tasks labeled \( \{1, 2, \ldots, n\} \) that need to be performed, and there are dependencies among them stipulating, for certain pairs \(i\) and \(j\), that \(i\) must be performed before \(j\). For example, the tasks may be courses, with prerequisite requirements stating that certain courses must be taken before others. Or the tasks may correspond to a pipeline of computing jobs, with assertions that the output of job \(i\) is used in determining the input to job \(j\), and hence job \(i\) must be done before job \(j\).

We can represent such an interdependent set of tasks by introducing a node for each task, and a directed edge \((i, j)\) whenever \(i\) must be done before \(j\). If the precedence relation is to be at all meaningful, the resulting graph \(G\) must be a DAG. Indeed, if it contained a cycle \(C\), there would be no way to do any of the tasks in \(C\): since each task in \(C\) cannot begin until some other one completes, no task in \(C\) could ever be done, since none could be done first.
3.6 Directed Acyclic Graphs and Topological Ordering

Let’s continue a little further with this picture of DAGs as precedence relations. Given a set of tasks with dependencies, it would be natural to seek a valid order in which the tasks could be performed, so that all dependencies are respected. Specifically, for a directed graph \( G \), we say that a **topological ordering** of \( G \) is an ordering of its nodes as \( v_1, v_2, \ldots, v_n \) so that for every edge \((v_i, v_j)\), we have \( i < j \). In other words, all edges point “forward” in the ordering. A topological ordering on tasks provides an order in which they can be safely performed; when we come to the task \( v_j \), all the tasks that are required to precede it have already been done. In Figure 3.7(b) we’ve labeled the nodes of the DAG from part (a) with a topological ordering; note that each edge indeed goes from a lower-indexed node to a higher-indexed node.

In fact, we can view a topological ordering of \( G \) as providing an immediate “proof” that \( G \) has no cycles, via the following.

**Proof.** Suppose, by way of contradiction, that \( G \) has a topological ordering \( v_1, v_2, \ldots, v_n \), and also has a cycle \( C \). Let \( v_i \) be the lowest-indexed node on \( C \), and let \( v_j \) be the node on \( C \) just before \( v_i \)—thus \((v_j, v_i)\) is an edge. But by our choice of \( i \), we have \( j > i \), which contradicts the assumption that \( v_1, v_2, \ldots, v_n \) was a topological ordering. ■

The proof of acyclicity that a topological ordering provides can be very useful, even visually. In Figure 3.7(c), we have drawn the same graph as in (a) and (b), but with the nodes laid out in the topological ordering. It is immediately clear that the graph in (c) is a DAG since each edge goes from left to right.

**Computing a Topological Ordering**  The main question we consider here is the converse of (3.18): Does every DAG have a topological ordering, and if so, how do we find one efficiently? A method to do this for every DAG would be very useful: it would show that for any precedence relation on a set of tasks without cycles, there is an efficiently computable order in which to perform the tasks.

**Designing and Analyzing the Algorithm**

In fact, the converse of (3.18) does hold, and we establish this via an efficient algorithm to compute a topological ordering. The key to this lies in finding a way to get started: which node do we put at the beginning of the topological ordering? Such a node \( v_1 \) would need to have no incoming edges, since any such incoming edge would violate the defining property of the topological
ordering, that all edges point forward. Thus, we need to prove the following fact.

**3.19** In every DAG $G$, there is a node $v$ with no incoming edges.

**Proof.** Let $G$ be a directed graph in which every node has at least one incoming edge. We show how to find a cycle in $G$; this will prove the claim. We pick any node $v$, and begin following edges backward from $v$: since $v$ has at least one incoming edge $(u, v)$, we can walk backward to $u$; then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$; and so on. We can continue this process indefinitely, since every node we encounter has an incoming edge. But after $n + 1$ steps, we will have visited some node $w$ twice. If we let $C$ denote the sequence of nodes encountered between successive visits to $w$, then clearly $C$ forms a cycle. ■

In fact, the existence of such a node $v$ is all we need to produce a topological ordering of $G$ by induction. Specifically, let us claim by induction that every DAG has a topological ordering. This is clearly true for DAGs on one or two nodes. Now suppose it is true for DAGs with up to some number of nodes $n$. Then, given a DAG $G$ on $n + 1$ nodes, we find a node $v$ with no incoming edges, as guaranteed by (3.19). We place $v$ first in the topological ordering; this is safe, since all edges out of $v$ will point forward. Now $G - \{v\}$ is a DAG, since deleting $v$ cannot create any cycles that weren’t there previously. Also, $G - \{v\}$ has $n$ nodes, so we can apply the induction hypothesis to obtain a topological ordering of $G - \{v\}$. We append the nodes of $G - \{v\}$ in this order after $v$; this is an ordering of $G$ in which all edges point forward, and hence it is a topological ordering.

Thus we have proved the desired converse of (3.18).

**3.20** If $G$ is a DAG, then $G$ has a topological ordering.

The inductive proof contains the following algorithm to compute a topological ordering of $G$.

To compute a topological ordering of $G$:
- Find a node $v$ with no incoming edges and order it first
- Delete $v$ from $G$
- Recursively compute a topological ordering of $G - \{v\}$
- and append this order after $v$

In Figure 3.8 we show the sequence of node deletions that occurs when this algorithm is applied to the graph in Figure 3.7. The shaded nodes in each iteration are those with no incoming edges; the crucial point, which is what
3.6 Directed Acyclic Graphs and Topological Ordering

Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm’s execution.

(3.19) guarantees, is that when we apply this algorithm to a DAG, there will always be at least one such node available to delete.

To bound the running time of this algorithm, we note that identifying a node \( v \) with no incoming edges, and deleting it from \( G \), can be done in \( O(n) \) time. Since the algorithm runs for \( n \) iterations, the total running time is \( O(n^2) \).

This is not a bad running time; and if \( G \) is very dense, containing \( \Theta(n^2) \) edges, then it is linear in the size of the input. But we may well want something better when the number of edges \( m \) is much less than \( n^2 \). In such a case, a running time of \( O(m + n) \) could be a significant improvement over \( \Theta(n^2) \).

In fact, we can achieve a running time of \( O(m + n) \) using the same high-level algorithm—iteratively deleting nodes with no incoming edges. We simply have to be more efficient in finding these nodes, and we do this as follows.

We declare a node to be “active” if it has not yet been deleted by the algorithm, and we explicitly maintain two things:

(a) for each node \( w \), the number of incoming edges that \( w \) has from active nodes; and
(b) the set \( S \) of all active nodes in \( G \) that have no incoming edges from other active nodes.
Chapter 3  Graphs

At the start, all nodes are active, so we can initialize (a) and (b) with a single pass through the nodes and edges. Then, each iteration consists of selecting a node \( v \) from the set \( S \) and deleting it. After deleting \( v \), we go through all nodes \( w \) to which \( v \) had an edge, and subtract one from the number of active incoming edges that we are maintaining for \( w \). If this causes the number of active incoming edges to \( w \) to drop to zero, then we add \( w \) to the set \( S \). Proceeding in this way, we keep track of nodes that are eligible for deletion at all times, while spending constant work per edge over the course of the whole algorithm.

**Solved Exercises**

**Solved Exercise 1**

Consider the directed acyclic graph \( G \) in Figure 3.9. How many topological orderings does it have?

**Solution**  Recall that a topological ordering of \( G \) is an ordering of the nodes as \( v_1, v_2, \ldots, v_n \) so that all edges point “forward”: for every edge \((v_i, v_j)\), we have \( i < j \).

So one way to answer this question would be to write down all \( 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120 \) possible orderings and check whether each is a topological ordering. But this would take a while.

Instead, we think about this as follows. As we saw in the text (or reasoning directly from the definition), the first node in a topological ordering must be one that has no edge coming into it. Analogously, the last node must be one that has no edge leaving it. Thus, in every topological ordering of \( G \), the node \( a \) must come first and the node \( e \) must come last.

Now we have to figure how the nodes \( b, c, \) and \( d \) can be arranged in the middle of the ordering. The edge \((c, d)\) enforces the requirement that \( c \) must come before \( d \); but \( b \) can be placed anywhere relative to these two: before both, between \( c \) and \( d \), or after both. This exhausts all the possibilities, and so we conclude that there are three possible topological orderings:

\[
\begin{align*}
  a, b, c, d, e \\
  a, c, b, d, e \\
  a, c, d, b, e
\end{align*}
\]

**Solved Exercise 2**

Some friends of yours are working on techniques for coordinating groups of mobile robots. Each robot has a radio transmitter that it uses to communicate
with a base station, and your friends find that if the robots get too close to one another, then there are problems with interference among the transmitters. So a natural problem arises: how to plan the motion of the robots in such a way that each robot gets to its intended destination, but in the process the robots don’t come close enough together to cause interference problems.

We can model this problem abstractly as follows. Suppose that we have an undirected graph \( G = (V, E) \), representing the floor plan of a building, and there are two robots initially located at nodes \( a \) and \( b \) in the graph. The robot at node \( a \) wants to travel to node \( c \) along a path in \( G \), and the robot at node \( b \) wants to travel to node \( d \). This is accomplished by means of a schedule: at each time step, the schedule specifies that one of the robots moves across a single edge, from one node to a neighboring node; at the end of the schedule, the robot from node \( a \) should be sitting on \( c \), and the robot from \( b \) should be sitting on \( d \).

A schedule is interference-free if there is no point at which the two robots occupy nodes that are at a distance \( \leq r \) from one another in the graph, for a given parameter \( r \). We’ll assume that the two starting nodes \( a \) and \( b \) are at a distance greater than \( r \), and so are the two ending nodes \( c \) and \( d \).

Give a polynomial-time algorithm that decides whether there exists an interference-free schedule by which each robot can get to its destination.

**Solution** This is a problem of the following general flavor. We have a set of possible configurations for the robots, where we define a configuration to be a choice of location for each one. We are trying to get from a given starting configuration \( (a, b) \) to a given ending configuration \( (c, d) \), subject to constraints on how we can move between configurations (we can only change one robot’s location to a neighboring node), and also subject to constraints on which configurations are “legal.”

This problem can be tricky to think about if we view things at the level of the underlying graph \( G \): for a given configuration of the robots—that is, the current location of each one—it’s not clear what rule we should be using to decide how to move one of the robots next. So instead we apply an idea that can be very useful for situations in which we’re trying to perform this type of search. We observe that our problem looks a lot like a path-finding problem, not in the original graph \( G \) but in the space of all possible configurations.

Let us define the following (larger) graph \( H \). The node set of \( H \) is the set of all possible configurations of the robots; that is, \( H \) consists of all possible pairs of nodes in \( G \). We join two nodes of \( H \) by an edge if they represent configurations that could be consecutive in a schedule; that is, \((u, v)\) and \((u', v')\) will be joined by an edge in \( H \) if one of the pairs \( u, u' \) or \( v, v' \) are equal, and the other pair corresponds to an edge in \( G \).
Chapter 3  Graphs

We can already observe that paths in $H$ from $(a, b)$ to $(c, d)$ correspond to schedules for the robots: such a path consists precisely of a sequence of configurations in which, at each step, one robot crosses a single edge in $G$. However, we have not yet encoded the notion that the schedule should be interference-free.

To do this, we simply delete from $H$ all nodes that correspond to configurations in which there would be interference. Thus we define $H'$ to be the graph obtained from $H$ by deleting all nodes $(u, v)$ for which the distance between $u$ and $v$ in $G$ is at most $r$.

The full algorithm is then as follows. We construct the graph $H'$, and then run the connectivity algorithm from the text to determine whether there is a path from $(a, b)$ to $(c, d)$. The correctness of the algorithm follows from the fact that paths in $H'$ correspond to schedules, and the nodes in $H'$ correspond precisely to the configurations in which there is no interference.

Finally, we need to consider the running time. Let $n$ denote the number of nodes in $G$, and $m$ denote the number of edges in $G$. We'll analyze the running time by doing three things: (1) bounding the size of $H'$ (which will in general be larger than $G$), (2) bounding the time it takes to construct $H'$, and (3) bounding the time it takes to search for a path from $(a, b)$ to $(c, d)$ in $H$.

1. First, then, let's consider the size of $H'$. $H'$ has at most $n^2$ nodes, since its nodes correspond to pairs of nodes in $G$. Now, how many edges does $H'$ have? A node $(u, v)$ will have edges to $(u', v)$ for each neighbor $u'$ of $u$ in $G$, and to $(u, v')$ for each neighbor $v'$ of $v$ in $G$. A simple upper bound says that there can be at most $n$ choices for $(u', v)$, and at most $n$ choices for $(u, v')$, so there are at most $2n$ edges incident to each node of $H'$. Summing over the (at most) $n^2$ nodes of $H'$, we have $O(n^3)$ edges.

   (We can actually give a better bound of $O(mn)$ on the number of edges in $H'$, by using the bound (3.9) we proved in Section 3.3 on the sum of the degrees in a graph. We'll leave this as a further exercise.)

2. Now we bound the time needed to construct $H'$. We first build $H$ by enumerating all pairs of nodes in $G$ in time $O(n^2)$, and constructing edges using the definition above in time $O(n)$ per node, for a total of $O(n^3)$. Now we need to figure out which nodes to delete from $H$ so as to produce $H'$. We can do this as follows. For each node $u$ in $G$, we run a breadth-first search from $u$ and identify all nodes $v$ within distance $r$ of $u$. We list all these pairs $(u, v)$ and delete them from $H$. Each breadth-first search in $G$ takes time $O(m + n)$, and we're doing one from each node, so the total time for this part is $O(mn + n^2)$.
3. Now we have $H'$, and so we just need to decide whether there is a path from $(a, b)$ to $(c, d)$. This can be done using the connectivity algorithm from the text in time that is linear in the number of nodes and edges of $H'$. Since $H'$ has $O(n^2)$ nodes and $O(n^3)$ edges, this final step takes polynomial time as well.

Exercises

1. Consider the directed acyclic graph $G$ in Figure 3.10. How many topological orderings does it have?

2. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with $n$ nodes and $m$ edges.

3. The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

   But suppose that we’re given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph $G$, it outputs one of two things: (a) a topological ordering, thus establishing that $G$ is a DAG; or (b) a cycle in $G$, thus establishing that $G$ is not a DAG. The running time of your algorithm should be $O(m + n)$ for a directed graph with $n$ nodes and $m$ edges.

4. Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they’ve caught—thanks to the fact that many species look very similar to one another.

   One day they return with $n$ butterflies, and they believe that each belongs to one of two different species, which we’ll call $A$ and $B$ for purposes of this discussion. They’d like to divide the $n$ specimens into two groups—those that belong to $A$ and those that belong to $B$—but it’s very hard for them to directly label any one specimen. So they decide to adopt the following approach.
For each pair of specimens $i$ and $j$, they study them carefully side by side. If they're confident enough in their judgment, then they label the pair $(i, j)$ either “same” (meaning they believe them both to come from the same species) or “different” (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair ambiguous.

So now they have the collection of $n$ specimens, as well as a collection of $m$ judgments (either “same” or “different”) for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species $A$ or $B$. So more concretely, we’ll declare the $m$ judgments to be consistent if it is possible to label each specimen either $A$ or $B$ in such a way that for each pair $(i, j)$ labeled “same,” it is the case that $i$ and $j$ have the same label; and for each pair $(i, j)$ labeled “different,” it is the case that $i$ and $j$ have different labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.

Give an algorithm with running time $O(m + n)$ that determines whether the $m$ judgments are consistent.

5. A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

6. We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at $u$, and obtain a tree $T$ that includes all nodes of $G$. Suppose we then compute a breadth-first search tree rooted at $u$, and obtain the same tree $T$. Prove that $G = T$. (In other words, if $T$ is both a depth-first search tree and a breadth-first search tree rooted at $u$, then $G$ cannot contain any edges that do not belong to $T$.)

7. Some friends of yours work on wireless networks, and they're currently studying the properties of a network of $n$ mobile devices. As the devices move around (actually, as their human owners move around), they define a graph at any point in time as follows: there is a node representing each of the $n$ devices, and there is an edge between device $i$ and device $j$ if the physical locations of $i$ and $j$ are no more than 500 meters apart. (If so, we say that $i$ and $j$ are "in range" of each other.)

They’d like it to be the case that the network of devices is connected at all times, and so they've constrained the motion of the devices to satisfy
the following property: at all times, each device \( i \) is within 500 meters of at least \( n/2 \) of the other devices. (We’ll assume \( n \) is an even number.) What they’d like to know is: Does this property by itself guarantee that the network will remain connected?

Here’s a concrete way to formulate the question as a claim about graphs.

Claim: Let \( G \) be a graph on \( n \) nodes, where \( n \) is an even number. If every node of \( G \) has degree at least \( n/2 \), then \( G \) is connected.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

8. A number of stories in the press about the structure of the Internet and the Web have focused on some version of the following question: How far apart are typical nodes in these networks? If you read these stories carefully, you find that many of them are confused about the difference between the diameter of a network and the average distance in a network; they often jump back and forth between these concepts as though they’re the same thing.

As in the text, we say that the distance between two nodes \( u \) and \( v \) in a graph \( G = (V, E) \) is the minimum number of edges in a path joining them; we’ll denote this by dist(\( u, v \)). We say that the diameter of \( G \) is the maximum distance between any pair of nodes; and we’ll denote this quantity by diam(\( G \)).

Let’s define a related quantity, which we’ll call the average pairwise distance in \( G \) (denoted \( apd(G) \)). We define \( apd(G) \) to be the average, over all \( \binom{n}{2} \) sets of two distinct nodes \( u \) and \( v \), of the distance between \( u \) and \( v \). That is,

\[
apd(G) = \left[ \frac{\sum_{[u,v] \subseteq V} \text{dist}(u,v)}{\binom{n}{2}} \right].\]

Here’s a simple example to convince yourself that there are graphs \( G \) for which \( \text{diam}(G) \neq \text{apd}(G) \). Let \( G \) be a graph with three nodes \( u, v, w \), and with the two edges \{\( u, v \}\} and \{\( v, w \}\}. Then

\[
\text{diam}(G) = \text{dist}(u, w) = 2,
\]

while

\[
apd(G) = \frac{\text{dist}(u, v) + \text{dist}(u, w) + \text{dist}(v, w)}{3} = \frac{4}{3}.
\]
Chapter 3  Graphs

Of course, these two numbers aren’t all that far apart in the case of this three-node graph, and so it’s natural to ask whether there’s always a close relation between them. Here’s a claim that tries to make this precise.

Claim: There exists a positive natural number $c$ so that for all connected graphs $G$, it is the case that

$$\frac{\text{diam}(G)}{\text{apd}(G)} \leq c.$$ 

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

9. There’s a natural intuition that two nodes that are far apart in a communication network—separated by many hops—have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here’s one that involves the susceptibility of paths to the deletion of nodes.

Suppose that an $n$-node undirected graph $G = (V, E)$ contains two nodes $s$ and $t$ such that the distance between $s$ and $t$ is strictly greater than $n/2$. Show that there must exist some node $v$, not equal to either $s$ or $t$, such that deleting $v$ from $G$ destroys all $s$-$t$ paths. (In other words, the graph obtained from $G$ by deleting $v$ contains no path from $s$ to $t$.) Give an algorithm with running time $O(m + n)$ to find such a node $v$.

10. A number of art museums around the country have been featuring work by an artist named Mark Lombardi (1951–2000), consisting of a set of intricately rendered graphs. Building on a great deal of research, these graphs encode the relationships among people involved in major political scandals over the past several decades: the nodes correspond to participants, and each edge indicates some type of relationship between a pair of participants. And so, if you peer closely enough at the drawings, you can trace out ominous-looking paths from a high-ranking U.S. government official, to a former business partner, to a bank in Switzerland, to a shadowy arms dealer.

Such pictures form striking examples of social networks, which, as we discussed in Section 3.1, have nodes representing people and organizations, and edges representing relationships of various kinds. And the short paths that abound in these networks have attracted considerable attention recently, as people ponder what they mean. In the case of Mark Lombardi’s graphs, they hint at the short set of steps that can carry you from the reputable to the disreputable.
Exercises

Of course, a single, spurious short path between nodes \( v \) and \( w \) in such a network may be more coincidental than anything else; a large number of short paths between \( v \) and \( w \) can be much more convincing. So in addition to the problem of computing a single shortest \( v \)-\( w \) path in a graph \( G \), social networks researchers have looked at the problem of determining the number of shortest \( v \)-\( w \) paths.

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph \( G = (V, E) \), and we identify two nodes \( v \) and \( w \) in \( G \). Give an algorithm that computes the number of shortest \( v \)-\( w \) paths in \( G \). (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be \( O(m + n) \) for a graph with \( n \) nodes and \( m \) edges.

11. You're helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are \( n \) computers in the system, labeled \( C_1, C_2, \ldots, C_n \), and as input you're given a collection of trace data indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples \( (C_i, C_j, t_k) \); such a triple indicates that \( C_i \) and \( C_j \) exchanged bits at time \( t_k \). There are \( m \) triples total.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing.

The security analysts you're working with would like to be able to answer questions of the following form: If the virus was inserted into computer \( C_a \) at time \( x \), could it possibly have infected computer \( C_b \) by time \( y \)? The mechanics of infection are simple: if an infected computer \( C_i \) communicates with an uninfected computer \( C_j \) at time \( t_k \) (in other words, if one of the triples \( (C_i, C_j, t_k) \) or \( (C_j, C_i, t_k) \) appears in the trace data), then computer \( C_j \) becomes infected as well, starting at time \( t_k \). Infection can thus spread from one machine to another across a sequence of communications, provided that no step in this sequence involves a move backward in time. Thus, for example, if \( C_i \) is infected by time \( t_k \), and the trace data contains triples \( (C_i, C_j, t_k) \) and \( (C_j, C_q, t_r) \), where \( t_k \leq t_r \), then \( C_q \) will become infected via \( C_j \). (Note that it is okay for \( t_k \) to be equal to \( t_r \); this would mean that \( C_j \) had open connections to both \( C_i \) and \( C_q \) at the same time, and so a virus could move from \( C_i \) to \( C_q \).)

For example, suppose \( n = 4 \), the trace data consists of the triples

\[(C_1, C_2, 4), \quad (C_2, C_4, 8), \quad (C_3, C_4, 8), \quad (C_1, C_4, 12),\]

For any query, contact us at help@tutorialsduniya.com
and the virus was inserted into computer $C_1$ at time 2. Then $C_3$ would be infected at time 8 by a sequence of three steps: first $C_2$ becomes infected at time 4, then $C_4$ gets the virus from $C_2$ at time 8, and then $C_3$ gets the virus from $C_4$ at time 8. On the other hand, if the trace data were

$$(C_2, C_3, 8), \ (C_1, C_4, 12), \ (C_1, C_2, 14),$$

and again the virus was inserted into computer $C_1$ at time 2, then $C_3$ would not become infected during the period of observation: although $C_2$ becomes infected at time 14, we see that $C_3$ only communicates with $C_2$ before $C_2$ was infected. There is no sequence of communications moving forward in time by which the virus could get from $C_1$ to $C_3$ in this second example.

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer $C_a$ at time $x$ could have infected computer $C_b$ by time $y$. The algorithm should run in time $O(m + n)$.

12. You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who've lived there over the past two hundred years.

From these interviews, they've learned about a set of $n$ people (all of them now deceased), whom we'll denote $P_1, P_2, \ldots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- For some $i$ and $j$, person $P_i$ died before person $P_j$ was born; or
- for some $i$ and $j$, the life spans of $P_i$ and $P_j$ overlapped at least partially.

Naturally, they're not sure that all these facts are correct; memories are not so good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the $n$ people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.
17 Amortized Analysis

In an *amortized analysis*, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 17.1 starts with aggregate analysis, in which we determine an upper bound $T(n)$ on the total cost of a sequence of $n$ operations. The average cost per operation is then $T(n)/n$. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 17.2 covers the accounting method, in which we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as “prepaid credit” on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 17.3 discusses the potential method, which is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the “potential energy” of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We shall use two examples to examine these three methods. One is a stack with the additional operation `MULTIPOP`, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation `INCREMENT`.
While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not—appear in the code. If, for example, we assign a credit to an object $x$ when using the accounting method, we have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design. In Section 17.4, for example, we shall use the potential method to analyze a dynamically expanding and contracting table.

17.1 Aggregate analysis

In **aggregate analysis**, we show that for all $n$, a sequence of $n$ operations takes worst-case time $T(n)$ in total. In the worst case, the average cost, or **amortized cost**, per operation is therefore $T(n)/n$. Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

**Stack operations**

In our first example of aggregate analysis, we analyze stacks that have been augmented with a new operation. Section 10.1 presented the two fundamental stack operations, each of which takes $O(1)$ time:

- $\text{PUSH}(S, x)$ pushes object $x$ onto stack $S$.
- $\text{POP}(S)$ pops the top of stack $S$ and returns the popped object. Calling POP on an empty stack generates an error.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of $n$ PUSH and POP operations is therefore $n$, and the actual running time for $n$ operations is therefore $\Theta(n)$.

Now we add the stack operation $\text{MULTIPOP}(S, k)$, which removes the $k$ top objects of stack $S$, popping the entire stack if the stack contains fewer than $k$ objects. Of course, we assume that $k$ is positive; otherwise the $\text{MULTIPOP}$ operation leaves the stack unchanged. In the following pseudocode, the operation $\text{STACK-EMPTY}$ returns TRUE if there are no objects currently on the stack, and FALSE otherwise.
17.1 Aggregate analysis

Figure 17.1 The action of MULTIPOP on a stack \( S \), shown initially in (a). The top 4 objects are popped by MULTIPOP\((S, 4)\), whose result is shown in (b). The next operation is MULTIPOP\((S, 7)\), which empties the stack—shown in (c)—since there were fewer than 7 objects remaining.

\[
\text{MULTIPOP}(S, k)
\]

1. while not STACK-EMPTY\((S)\) and \( k > 0 \)
2. \( \text{POP}(S) \)
3. \( k = k - 1 \)

Figure 17.1 shows an example of MULTIPOP.

What is the running time of \( \text{MULTIPOP}(S, k) \) on a stack of \( s \) objects? The actual running time is linear in the number of \text{POP} operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for \text{PUSH} and \text{POP}. The number of iterations of the while loop is the number \( \min(s, k) \) of objects popped off the stack. Each iteration of the loop makes one call to \text{POP} in line 2. Thus, the total cost of \text{MULTIPOP} is \( \min(s, k) \), and the actual running time is a linear function of this cost.

Let us analyze a sequence of \( n \) \text{PUSH}, \text{POP}, and \text{MULTIPOP} operations on an initially empty stack. The worst-case cost of a \text{MULTIPOP} operation in the sequence is \( O(n) \), since the stack size is at most \( n \). The worst-case time of any stack operation is therefore \( O(n) \), and hence a sequence of \( n \) operations costs \( O(n^2) \), since we may have \( O(n) \) \text{MULTIPOP} operations costing \( O(n) \) each. Although this analysis is correct, the \( O(n^2) \) result, which we obtained by considering the worst-case cost of each operation individually, is not tight.

Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of \( n \) operations. In fact, although a single \text{MULTIPOP} operation can be expensive, any sequence of \( n \) \text{PUSH}, \text{POP}, and \text{MULTIPOP} operations on an initially empty stack can cost at most \( O(n) \). Why? We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that \text{POP} can be called on a nonempty stack, including calls within \text{MULTIPOP}, is at most the number of \text{PUSH} operations, which is at most \( n \). For any value of \( n \), any sequence of \( n \) \text{PUSH}, \text{POP}, and \text{MULTIPOP} operations takes a total of \( O(n) \) time. The average cost of an operation is \( O(n)/n = O(1) \). In aggregate

For any query, contact us at help@tutorialsduniya.com
analysis, we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

We emphasize again that although we have just shown that the average cost, and hence the running time, of a stack operation is $O(1)$, we did not use probabilistic reasoning. We actually showed a worst-case bound of $O(n)$ on a sequence of $n$ operations. Dividing this total cost by $n$ yielded the average cost per operation, or the amortized cost.

**Incrementing a binary counter**

As another example of aggregate analysis, consider the problem of implementing a $k$-bit binary counter that counts upward from 0. We use an array $A[0..k-1]$ of bits, where $A.length = k$, as the counter. A binary number $x$ that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$. To add 1 (modulo $2^k$) to the value in the counter, we use the following procedure.

```
INCREMENT (A)
1  i = 0
2  while i < A.length and A[i] == 1
3     A[i] = 0
4     i = i + 1
5  if i < A.length
6     A[i] = 1
```

Figure 17.2 shows what happens to a binary counter as we increment it 16 times, starting with the initial value 0 and ending with the value 16. At the start of each iteration of the while loop in lines 2–4, we wish to add a 1 into position $i$. If $A[i] = 1$, then adding 1 flips the bit to 0 in position $i$ and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that line 6 adds a 1 into position $i$, flipping the 0 to a 1. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, in which array $A$ contains all 1s. Thus, a sequence of $n$ INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

We can tighten our analysis to yield a worst-case cost of $O(n)$ for a sequence of $n$ INCREMENT operations by observing that not all bits flip each time INCREMENT is called. As Figure 17.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of $n$ INCREMENT
### 17.1 Aggregate analysis

#### Figure 17.2
An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.

Operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of $n$ INCREMENT operations. In general, for $i = 0, 1, \ldots, k - 1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of $n$ INCREMENT operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number of flips in the sequence is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n,$$

by equation (A.6). The worst-case time for a sequence of $n$ INCREMENT operations on an initially zero counter is therefore $O(n)$. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$. 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
</tbody>
</table>
Exercises

17.1-1
If the set of stack operations included a MULTIPUSH operation, which pushes $k$ items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

17.1-2
Show that if a DECREMENT operation were included in the $k$-bit counter example, $n$ operations could cost as much as $\Theta(nk)$ time.

17.1-3
Suppose we perform a sequence of $n$ operations on a data structure in which the $i$th operation costs $i$ if $i$ is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.
17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform, or FFT. The first step of the FFT algorithm performs a bit-reversal permutation on an input array \( A[0..n-1] \) whose length is \( n = 2^k \) for some nonnegative integer \( k \). This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index \( a \) as a \( k \)-bit sequence \( \langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle \), where \( a = \sum_{i=0}^{k-1} a_i 2^i \). We define

\[
\text{rev}_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle;
\]

thus,

\[
\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.
\]

For example, if \( n = 16 \) (or, equivalently, \( k = 4 \)), then \( \text{rev}_k(3) = 12 \), since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12.

(a) Given a function \( \text{rev}_k \) that runs in \( \Theta(k) \) time, write an algorithm to perform the bit-reversal permutation on an array of length \( n = 2^k \) in \( O(nk) \) time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a “bit-reversed counter” and a procedure \( \text{BIT-REVERSED-INCREMENT} \) that, when given a bit-reversed-counter value \( a \), produces \( \text{rev}_k(\text{rev}_k(a) + 1) \). If \( k = 4 \), for example, and the bit-reversed counter starts at 0, then successive calls to \( \text{BIT-REVERSED-INCREMENT} \) produce the sequence

\( 0000, 1000, 0100, 1100, 0010, 1010, \ldots = 0, 8, 4, 12, 2, 10, \ldots \).

(b) Assume that the words in your computer store \( k \)-bit values and that in unit time, your computer can manipulate the binary values with operations such as shifting left or right by arbitrary amounts, bitwise-AND, bitwise-OR, etc. Describe an implementation of the \( \text{BIT-REVERSED-INCREMENT} \) procedure that allows the bit-reversal permutation on an \( n \)-element array to be performed in a total of \( O(n) \) time.

(c) Suppose that you can shift a word left or right by only one bit in unit time. Is it still possible to implement an \( O(n) \)-time bit-reversal permutation?
Problems for Chapter 17

17-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of \( n \) elements. Let \( k = \lceil \log(n + 1) \rceil \), and let the binary representation of \( n \) be \( \langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle \). We have \( k \) sorted arrays \( A_0, A_1, \ldots, A_{k-1} \), where for \( i = 0, 1, \ldots, k - 1 \), the length of array \( A_i \) is \( 2^i \). Each array is either full or empty, depending on whether \( n_i = 1 \) or \( n_i = 0 \), respectively. The total number of elements held in all \( k \) arrays is therefore \( \sum_{i=0}^{k-1} n_i 2^i = n \). Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

\( a. \) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

\( b. \) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

\( c. \) Discuss how to implement DELETE.

17-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node \( x \) the attribute \( x.size \) giving the number of keys stored in the subtree rooted at \( x \). Let \( \alpha \) be a constant in the range \( 1/2 \leq \alpha < 1 \). We say that a given node \( x \) is \( \alpha \)-balanced if \( x.left.size \leq \alpha \cdot x.size \) and \( x.right.size \leq \alpha \cdot x.size \). The tree as a whole is \( \alpha \)-balanced if every node in the tree is \( \alpha \)-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

\( a. \) A \( 1/2 \)-balanced tree is, in a sense, as balanced as it can be. Given a node \( x \) in an arbitrary binary search tree, show how to rebuild the subtree rooted at \( x \) so that it becomes \( 1/2 \)-balanced. Your algorithm should run in time \( \Theta(x.size) \), and it can use \( O(x.size) \) auxiliary storage.

\( b. \) Show that performing a search in an \( n \)-node \( \alpha \)-balanced binary search tree takes \( O(\log n) \) worst-case time.

For the remainder of this problem, assume that the constant \( \alpha \) is strictly greater than \( 1/2 \). Suppose that we implement INSERT and DELETE as usual for an \( n \)-node binary search tree, except that after every such operation, if any node in the tree is no longer \( \alpha \)-balanced, then we “rebuild” the subtree rooted at the highest such node in the tree so that it becomes \( 1/2 \)-balanced.
Chapter 17  Amortized Analysis

We shall analyze this rebuilding scheme using the potential method. For a node \( x \) in a binary search tree \( T \), we define
\[
\Delta(x) = |x\text{.left.size} - x\text{.right.size}|
\]
and we define the potential of \( T \) as
\[
\Phi(T) = c \sum_{x \in T: \Delta(x) \geq 2} \Delta(x),
\]
where \( c \) is a sufficiently large constant that depends on \( \alpha \).

c. Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

d. Suppose that \( m \) units of potential can pay for rebuilding an \( m \)-node subtree. How large must \( c \) be in terms of \( \alpha \) in order for it to take \( O(1) \) amortized time to rebuild a subtree that is not \( \alpha \)-balanced?

e. Show that inserting a node into or deleting a node from an \( n \)-node \( \alpha \)-balanced tree costs \( O(\lg n) \) amortized time.

17-4  The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform structural modifications: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only \( O(1) \) rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

a. Describe a legal red-black tree with \( n \) nodes such that calling RB-INSERT to add the \((n + 1)\)st node causes \( \Omega(\lg n) \) color changes. Then describe a legal red-black tree with \( n \) nodes for which calling RB-DELETE on a particular node causes \( \Omega(\lg n) \) color changes.

Although the worst-case number of color changes per operation can be logarithmic, we shall prove that any sequence of \( m \) RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes \( O(m) \) structural modifications in the worst case. Note that we count each color change as a structural modification.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are terminating: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (Hint: Look at Figures 13.5, 13.6, and 13.7.)
We shall first analyze the structural modifications when only insertions are performed. Let $T$ be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in $T$. Assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

c. Let $T'$ be the result of applying Case 1 of RB-INSERT-FIXUP to $T$. Argue that $\Phi(T') = \Phi(T) - 1$.

d. When we insert a node into a red-black tree using RB-INSERT, we can break the operation into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node $x$,

$$w(x) = \begin{cases} 
0 & \text{if } x \text{ is red}, \\
1 & \text{if } x \text{ is black and has no red children}, \\
0 & \text{if } x \text{ is black and has one red child}, \\
2 & \text{if } x \text{ is black and has two red children}.
\end{cases}$$

Now we redefine the potential of a red-black tree $T$ as

$$\Phi(T) = \sum_{x \in T} w(x),$$

and let $T'$ be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to $T$.

f. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

g. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

h. Complete the proof that in the worst case, any sequence of $m$ RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.
17-5 Competitive analysis of self-organizing lists with move-to-front

A self-organizing list is a linked list of $n$ elements, in which each element has a unique key. When we search for an element in the list, we are given a key, and we want to find an element with that key.

A self-organizing list has two important properties:

1. To find an element in the list, given its key, we must traverse the list from the beginning until we encounter the element with the given key. If that element is the $k$th element from the start of the list, then the cost to find the element is $k$.

2. We may reorder the list elements after any operation, according to a given rule with a given cost. We may choose any heuristic we like to decide how to reorder the list.

Assume that we start with a given list of $n$ elements, and we are given an access sequence $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle$ of keys to find, in order. The cost of the sequence is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this problem focuses on transposing adjacent list elements—switching their positions in the list—with a unit cost for each transpose operation. You will show, by means of a potential function, that a particular heuristic for reordering the list, move-to-front, entails a total cost no worse than 4 times that of any other heuristic for maintaining the list order—even if the other heuristic knows the access sequence in advance! We call this type of analysis a competitive analysis.

For a heuristic $H$ and a given initial ordering of the list, denote the access cost of sequence $\sigma$ by $C_H(\sigma)$. Let $m$ be the number of accesses in $\sigma$.

a. Argue that if heuristic $H$ does not know the access sequence in advance, then the worst-case cost for $H$ on an access sequence $\sigma$ is $C_H(\sigma) = \Omega(mn)$.

With the move-to-front heuristic, immediately after searching for an element $x$, we move $x$ to the first position on the list (i.e., the front of the list).

Let $\text{rank}_L(x)$ denote the rank of element $x$ in list $L$, that is, the position of $x$ in list $L$. For example, if $x$ is the fourth element in $L$, then $\text{rank}_L(x) = 4$. Let $c_i$ denote the cost of access $\sigma_i$ using the move-to-front heuristic, which includes the cost of finding the element in the list and the cost of moving it to the front of the list by a series of transpositions of adjacent list elements.

b. Show that if $\sigma_i$ accesses element $x$ in list $L$ using the move-to-front heuristic, then $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Now we compare move-to-front with any other heuristic $H$ that processes an access sequence according to the two properties above. Heuristic $H$ may transpose
elements in the list in any way it wants, and it might even know the entire access sequence in advance.

Let $L_i$ be the list after access $\sigma_i$ using move-to-front, and let $L_i^*$ be the list after access $\sigma_i$ using heuristic $H$. We denote the cost of access $\sigma_i$ by $c_i$ for move-to-front and by $c_i^*$ for heuristic $H$. Suppose that heuristic $H$ performs $t_i^*$ transpositions during access $\sigma_i$.

c. In part (b), you showed that $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Now show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$.

We define an inversion in list $L_i$ as a pair of elements $y$ and $z$ such that $y$ precedes $z$ in $L_i$ and $z$ precedes $y$ in list $L_i^*$. Suppose that list $L_i$ has $q_i$ inversions after processing the access sequence $\langle \sigma_1, \sigma_2, \ldots, \sigma_i \rangle$. Then, we define a potential function $\Phi$ that maps $L_i$ to a real number by $\Phi(L_i) = 2q_i$. For example, if $L_i$ has the elements $\langle e, c, a, d, b \rangle$ and $L_i^*$ has the elements $\langle c, a, b, d, e \rangle$, then $L_i$ has 5 inversions $\langle (e, c), (e, a), (e, d), (e, b), (d, b) \rangle$, and so $\Phi(L_i) = 10$. Observe that $\Phi(L_i) \geq 0$ for all $i$ and that, if move-to-front and heuristic $H$ start with the same list $L_0$, then $\Phi(L_0) = 0$.

d. Argue that a transposition either increases the potential by 2 or decreases the potential by 2.

Suppose that access $\sigma_i$ finds the element $x$. To understand how the potential changes due to $\sigma_i$, let us partition the elements other than $x$ into four sets, depending on where they are in the lists just before the $i$th access:

- Set $A$ consists of elements that precede $x$ in both $L_{i-1}$ and $L_{i-1}^*$.
- Set $B$ consists of elements that precede $x$ in $L_{i-1}$ and follow $x$ in $L_{i-1}^*$.
- Set $C$ consists of elements that follow $x$ in $L_{i-1}$ and precede $x$ in $L_{i-1}^*$.
- Set $D$ consists of elements that follow $x$ in both $L_{i-1}$ and $L_{i-1}^*$.

e. Argue that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ and $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

f. Show that access $\sigma_i$ causes a change in potential of

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*),$$

where, as before, heuristic $H$ performs $t_i^*$ transpositions during access $\sigma_i$.

Define the amortized cost $\hat{c}_i$ of access $\sigma_i$ by $\hat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$.

g. Show that the amortized cost $\hat{c}_i$ of access $\sigma_i$ is bounded from above by $4c_i^*$.

h. Conclude that the cost $C_{\text{MTF}}(\sigma)$ of access sequence $\sigma$ with move-to-front is at most 4 times the cost $C_H(\sigma)$ of $\sigma$ with any other heuristic $H$, assuming that both heuristics start with the same list.
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.</td>
<td>Greedy Technique</td>
<td>Ch – 4.1, 4.2, 4.4, 4.5 (excluding reverse delete algorithm), 4.6 [3]</td>
</tr>
</tbody>
</table>

**TutorialsDuniya.com**

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

 Compiled Books: https://www.tutorialsduniya.com/compiled-books
 Programs: https://www.tutorialsduniya.com/programs
 Question Papers: https://www.tutorialsduniya.com/question-papers
 Python Notes: https://www.tutorialsduniya.com/python
 Java Notes: https://www.tutorialsduniya.com/java
 JavaScript Notes: https://www.tutorialsduniya.com/javascript
 JSP Notes: https://www.tutorialsduniya.com/jsp
 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
 OR Notes: https://www.tutorialsduniya.com/operational-research
Chapter 4

Greedy Algorithms

In *Wall Street*, that iconic movie of the 1980s, Michael Douglas gets up in front of a room full of stockholders and proclaims, “Greed...is good. Greed is right. Greed works.” In this chapter, we’ll be taking a much more understated perspective as we investigate the pros and cons of short-sighted greed in the design of algorithms. Indeed, our aim is to approach a number of different computational problems with a recurring set of questions: Is greed good? Does greed work?

It is hard, if not impossible, to define precisely what is meant by a greedy algorithm. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

When a greedy algorithm succeeds in solving a nontrivial problem optimally, it typically implies something interesting and useful about the structure of the problem itself; there is a local decision rule that one can use to construct optimal solutions. And as we’ll see later, in Chapter 11, the same is true of problems in which a greedy algorithm can produce a solution that is guaranteed to be close to optimal, even if it does not achieve the precise optimum. These are the kinds of issues we’ll be dealing with in this chapter. It’s easy to invent greedy algorithms for almost any problem; finding cases in which they work well, and proving that they work well, is the interesting challenge.

The first two sections of this chapter will develop two basic methods for proving that a greedy algorithm produces an optimal solution to a problem. One can view the first approach as establishing that the greedy algorithm stays ahead. By this we mean that if one measures the greedy algorithm’s progress
in a step-by-step fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution. The second approach is known as an exchange argument, and it is more general: one considers any possible solution to the problem and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will follow that the greedy algorithm must have found a solution that is at least as good as any other solution.

Following our introduction of these two styles of analysis, we focus on several of the most well-known applications of greedy algorithms: shortest paths in a graph, the Minimum Spanning Tree Problem, and the construction of Huffman codes for performing data compression. They each provide nice examples of our analysis techniques. We also explore an interesting relationship between minimum spanning trees and the long-studied problem of clustering. Finally, we consider a more complex application, the Minimum-Cost Arborescence Problem, which further extends our notion of what a greedy algorithm is.

### 4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead

Let’s recall the Interval Scheduling Problem, which was the first of the five representative problems we considered in Chapter 1. We have a set of requests \( \{1, 2, \ldots, n\} \); the \( i \)th request corresponds to an interval of time starting at \( s(i) \) and finishing at \( f(i) \). (Note that we are slightly changing the notation from Section 1.2, where we used \( s_i \) rather than \( s(i) \) and \( f_i \) rather than \( f(i) \). This change of notation will make things easier to talk about in the proofs.) We’ll say that a subset of the requests is compatible if no two of them overlap in time, and our goal is to accept as large a compatible subset as possible. Compatible sets of maximum size will be called optimal.

**Designing a Greedy Algorithm**

Using the Interval Scheduling Problem, we can make our discussion of greedy algorithms much more concrete. The basic idea in a greedy algorithm for interval scheduling is to use a simple rule to select a first request \( i_1 \). Once a request \( i_1 \) is accepted, we reject all requests that are not compatible with \( i_1 \). We then select the next request \( i_2 \) to be accepted, and again reject all requests that are not compatible with \( i_2 \). We continue in this fashion until we run out of requests. The challenge in designing a good greedy algorithm is in deciding which simple rule to use for the selection—and there are many natural rules for this problem that do not give good solutions.

Let’s try to think of some of the most natural rules and see how they work.
The most obvious rule might be to always select the available request that starts earliest—that is, the one with minimal start time \( s(i) \). This way our resource starts being used as quickly as possible.

This method does not yield an optimal solution. If the earliest request \( i \) is for a very long interval, then by accepting request \( i \) we may have to reject a lot of requests for shorter time intervals. Since our goal is to satisfy as many requests as possible, we will end up with a suboptimal solution. In a really bad case—say, when the finish time \( f(i) \) is the maximum among all requests—the accepted request \( i \) keeps our resource occupied for the whole time. In this case our greedy method would accept a single request, while the optimal solution could accept many. Such a situation is depicted in Figure 4.1(a).

This might suggest that we should start out by accepting the request that requires the smallest interval of time—namely, the request for which \( f(i) - s(i) \) is as small as possible. As it turns out, this is a somewhat better rule than the previous one, but it still can produce a suboptimal schedule. For example, in Figure 4.1(b), accepting the short interval in the middle would prevent us from accepting the other two, which form an optimal solution.

![Figure 4.1](https://www.tutorialsduniya.com) Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.
Chapter 4  Greedy Algorithms

- In the previous greedy rule, our problem was that the second request competes with both the first and the third—that is, accepting this request made us reject two other requests. We could design a greedy algorithm that is based on this idea: for each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of noncompatible requests. (In other words, we select the interval with the fewest “conflicts.”) This greedy choice would lead to the optimum solution in the previous example. In fact, it is quite a bit harder to design a bad example for this rule; but it can be done, and we’ve drawn an example in Figure 4.1(c). The unique optimal solution in this example is to accept the four requests in the top row. The greedy method suggested here accepts the middle request in the second row and thereby ensures a solution of size no greater than three.

A greedy rule that does lead to the optimal solution is based on a fourth idea: we should accept first the request that finishes first, that is, the request \( i \) for which \( f(i) \) is as small as possible. This is also quite a natural idea: we ensure that our resource becomes free as soon as possible while still satisfying one request. In this way we can maximize the time left to satisfy other requests.

Let us state the algorithm a bit more formally. We will use \( R \) to denote the set of requests that we have neither accepted nor rejected yet, and use \( A \) to denote the set of accepted requests. For an example of how the algorithm runs, see Figure 4.2.

```
Initially let \( R \) be the set of all requests, and let \( A \) be empty
While \( R \) is not yet empty
    Choose a request \( i \in R \) that has the smallest finishing time
    Add request \( i \) to \( A \)
    Delete all requests from \( R \) that are not compatible with request \( i \)
EndWhile
Return the set \( A \) as the set of accepted requests
```

\( \Rightarrow \) **Analyzing the Algorithm**

While this greedy method is quite natural, it is certainly not obvious that it returns an optimal set of intervals. Indeed, it would only be sensible to reserve judgment on its optimality: the ideas that led to the previous nonoptimal versions of the greedy method also seemed promising at first.

As a start, we can immediately declare that the intervals in the set \( A \) returned by the algorithm are all compatible.

\[(4.1) \quad A \text{ is a compatible set of requests.}\]
4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead

Intervals numbered in order

Selecting interval 1

Selecting interval 3

Selecting interval 5

Selecting interval 8

Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

What we need to show is that this solution is optimal. So, for purposes of comparison, let \( \mathcal{O} \) be an optimal set of intervals. Ideally one might want to show that \( A = \mathcal{O} \), but this is too much to ask: there may be many optimal solutions, and at best \( A \) is equal to a single one of them. So instead we will simply show that \( |A| = |\mathcal{O}| \), that is, that \( A \) contains the same number of intervals as \( \mathcal{O} \) and hence is also an optimal solution.

The idea underlying the proof, as we suggested initially, will be to find a sense in which our greedy algorithm “stays ahead” of this solution \( \mathcal{O} \). We will compare the partial solutions that the greedy algorithm constructs to initial segments of the solution \( \mathcal{O} \), and show that the greedy algorithm is doing better in a step-by-step fashion.

We introduce some notation to help with this proof. Let \( i_1, \ldots, i_k \) be the set of requests in \( A \) in the order they were added to \( A \). Note that \( |A| = k \). Similarly, let the set of requests in \( \mathcal{O} \) be denoted by \( j_1, \ldots, j_m \). Our goal is to prove that \( k = m \). Assume that the requests in \( \mathcal{O} \) are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in \( \mathcal{O} \) are compatible, which implies that the start points have the same order as the finish points.
Our intuition for the greedy method came from wanting our resource to become free again as soon as possible after satisfying the first request. And indeed, our greedy rule guarantees that \( f(i_1) \leq f(j_1) \). This is the sense in which we want to show that our greedy rule “stays ahead”—that each of its intervals finishes at least as soon as the corresponding interval in the set \( \mathcal{O} \). Thus we now prove that for each \( r \geq 1 \), the \( r \)th accepted request in the algorithm’s schedule finishes no later than the \( r \)th request in the optimal schedule.

\[(4.2) \quad \text{For all indices } r \leq k \text{ we have } f(i_r) \leq f(j_r).\]

**Proof.** We will prove this statement by induction. For \( r = 1 \) the statement is clearly true: the algorithm starts by selecting the request \( i_1 \) with minimum finish time.

Now let \( r > 1 \). We will assume as our induction hypothesis that the statement is true for \( r - 1 \), and we will try to prove it for \( r \). As shown in Figure 4.3, the induction hypothesis lets us assume that \( f(i_{r-1}) \leq f(j_{r-1}) \). In order for the algorithm’s \( r \)th interval not to finish earlier as well, it would need to “fall behind” as shown. But there’s a simple reason why this could not happen: rather than choose a later-finishing interval, the greedy algorithm always has the option (at worst) of choosing \( j_r \) and thus fulfilling the induction step.

We can make this argument precise as follows. We know (since \( \mathcal{O} \) consists of compatible intervals) that \( f(j_{r-1}) \leq s(j_r) \). Combining this with the induction hypothesis \( f(i_{r-1}) \leq f(j_{r-1}) \), we get \( f(i_{r-1}) \leq s(j_r) \). Thus the interval \( j_r \) is in the set \( R \) of available intervals at the time when the greedy algorithm selects \( i_r \). The greedy algorithm selects the available interval with smallest finish time; since interval \( j_r \) is one of these available intervals, we have \( f(i_r) \leq f(j_r) \). This completes the induction step. \( \square \)

Thus we have formalized the sense in which the greedy algorithm is remaining ahead of \( \mathcal{O} \): for each \( r \), the \( r \)th interval it selects finishes at least as soon as the \( r \)th interval in \( \mathcal{O} \). We now see why this implies the optimality of the greedy algorithm’s set \( A \).
4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead

(4.3) **The greedy algorithm returns an optimal set A.**

**Proof.** We will prove the statement by contradiction. If $A$ is not optimal, then an optimal set $\emptyset$ must have more requests, that is, we must have $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request $j_{k+1}$ in $\emptyset$. This request starts after request $j_k$ ends, and hence after $i_k$ ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests $R$ still contains $j_{k+1}$. But the greedy algorithm stops with request $i_k$, and it is only supposed to stop when $R$ is empty—a contradiction. ■

**Implementation and Running Time** We can make our algorithm run in time $O(n \log n)$ as follows. We begin by sorting the $n$ requests in order of finishing time and labeling them in this order; that is, we will assume that $f(i) \leq f(j)$ when $i < j$. This takes time $O(n \log n)$. In an additional $O(n)$ time, we construct an array $S[1 \ldots n]$ with the property that $S[i]$ contains the value $s(i)$.

We now select requests by processing the intervals in order of increasing $f(i)$. We always select the first interval; we then iterate through the intervals in order until reaching the first interval $j$ for which $s(j) \geq f(1)$; we then select this one as well. More generally, if the most recent interval we’ve selected ends at time $f$, we continue iterating through subsequent intervals until we reach the first $j$ for which $s(j) \geq f$. In this way, we implement the greedy algorithm analyzed above in one pass through the intervals, spending constant time per interval. Thus this part of the algorithm takes time $O(n)$.

**Extensions**

The Interval Scheduling Problem we considered here is a quite simple scheduling problem. There are many further complications that could arise in practical settings. The following point out issues that we will see later in the book in various forms.

- In defining the problem, we assumed that all requests were known to the scheduling algorithm when it was choosing the compatible subset. It would also be natural, of course, to think about the version of the problem in which the scheduler needs to make decisions about accepting or rejecting certain requests before knowing about the full set of requests. Customers (requestors) may well be impatient, and they may give up and leave if the scheduler waits too long to gather information about all other requests. An active area of research is concerned with such online algorithms, which must make decisions as time proceeds, without knowledge of future input.
Our goal was to maximize the number of satisfied requests. But we could picture a situation in which each request has a different value to us. For example, each request $i$ could also have a value $v_i$ (the amount gained by satisfying request $i$), and the goal would be to maximize our income: the sum of the values of all satisfied requests. This leads to the Weighted Interval Scheduling Problem, the second of the representative problems we described in Chapter 1.

There are many other variants and combinations that can arise. We now discuss one of these further variants in more detail, since it forms another case in which a greedy algorithm can be used to produce an optimal solution.

A Related Problem: Scheduling All Intervals

The Problem In the Interval Scheduling Problem, there is a single resource and many requests in the form of time intervals, so we must choose which requests to accept and which to reject. A related problem arises if we have many identical resources available and we wish to schedule all the requests using as few resources as possible. Because the goal here is to partition all intervals across multiple resources, we will refer to this as the Interval Partitioning Problem.\(^1\)

For example, suppose that each request corresponds to a lecture that needs to be scheduled in a classroom for a particular interval of time. We wish to satisfy all these requests, using as few classrooms as possible. The classrooms at our disposal are thus the multiple resources, and the basic constraint is that any two lectures that overlap in time must be scheduled in different classrooms. Equivalently, the interval requests could be jobs that need to be processed for a specific period of time, and the resources are machines capable of handling these jobs. Much later in the book, in Chapter 10, we will see a different application of this problem in which the intervals are routing requests that need to be allocated bandwidth on a fiber-optic cable.

As an illustration of the problem, consider the sample instance in Figure 4.4(a). The requests in this example can all be scheduled using three resources; this is indicated in Figure 4.4(b), where the requests are rearranged into three rows, each containing a set of nonoverlapping intervals. In general, one can imagine a solution using $k$ resources as a rearrangement of the requests into $k$ rows of nonoverlapping intervals: the first row contains all the intervals

\(^1\) The problem is also referred to as the Interval Coloring Problem; the terminology arises from thinking of the different resources as having distinct colors—all the intervals assigned to a particular resource are given the corresponding color.
4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead

Figure 4.4 (a) An instance of the Interval Partitioning Problem with ten intervals (a through j). (b) A solution in which all intervals are scheduled using three resources: each row represents a set of intervals that can all be scheduled on a single resource.

assigned to the first resource, the second row contains all those assigned to the second resource, and so forth.

Now, is there any hope of using just two resources in this sample instance? Clearly the answer is no. We need at least three resources since, for example, intervals a, b, and c all pass over a common point on the time-line, and hence they all need to be scheduled on different resources. In fact, one can make this last argument in general for any instance of Interval Partitioning. Suppose we define the depth of a set of intervals to be the maximum number that pass over any single point on the time-line. Then we claim

(4.4) In any instance of Interval Partitioning, the number of resources needed is at least the depth of the set of intervals.

Proof. Suppose a set of intervals has depth d, and let $I_1, \ldots, I_d$ all pass over a common point on the time-line. Then each of these intervals must be scheduled on a different resource, so the whole instance needs at least d resources. ■

We now consider two questions, which turn out to be closely related. First, can we design an efficient algorithm that schedules all intervals using the minimum possible number of resources? Second, is there always a schedule using a number of resources that is equal to the depth? In effect, a positive answer to this second question would say that the only obstacles to partitioning intervals are purely local—a set of intervals all piled over the same point. It’s not immediately clear that there couldn’t exist other, “long-range” obstacles that push the number of required resources even higher.
We now design a simple greedy algorithm that schedules all intervals using a number of resources equal to the depth. This immediately implies the optimality of the algorithm: in view of (4.4), no solution could use a number of resources that is smaller than the depth. The analysis of our algorithm will therefore illustrate another general approach to proving optimality: one finds a simple, “structural” bound asserting that every possible solution must have at least a certain value, and then one shows that the algorithm under consideration always achieves this bound.

**Designing the Algorithm** Let \( d \) be the depth of the set of intervals; we show how to assign a label to each interval, where the labels come from the set of numbers \( \{1, 2, \ldots, d\} \), and the assignment has the property that overlapping intervals are labeled with different numbers. This gives the desired solution, since we can interpret each number as the name of a resource, and the label of each interval as the name of the resource to which it is assigned.

The algorithm we use for this is a simple one-pass greedy strategy that orders intervals by their starting times. We go through the intervals in this order, and try to assign to each interval we encounter a label that hasn’t already been assigned to any previous interval that overlaps it. Specifically, we have the following description.

\[
\begin{align*}
&\text{Sort the intervals by their start times, breaking ties arbitrarily} \\
&\text{Let } I_1, I_2, \ldots, I_n \text{ denote the intervals in this order} \\
&\text{For } j = 1, 2, 3, \ldots, n \\
&\quad \text{For each interval } I_i \text{ that precedes } I_j \text{ in sorted order and overlaps it} \\
&\quad \quad \text{Exclude the label of } I_i \text{ from consideration for } I_j \\
&\quad \text{Endfor} \\
&\quad \text{If there is any label from } \{1, 2, \ldots, d\} \text{ that has not been excluded then} \\
&\quad \quad \text{Assign a nonexcluded label to } I_j \\
&\quad \text{Else} \\
&\quad \quad \text{Leave } I_j \text{ unlabeled} \\
&\quad \text{Endif} \\
&\text{Endfor}
\end{align*}
\]

**Analyzing the Algorithm** We claim the following.

(4.5) If we use the greedy algorithm above, every interval will be assigned a label, and no two overlapping intervals will receive the same label.

**Proof.** First let’s argue that no interval ends up unlabeled. Consider one of the intervals \( I_j \), and suppose there are \( t \) intervals earlier in the sorted order that overlap it. These \( t \) intervals, together with \( I_j \), form a set of \( t + 1 \) intervals that all pass over a common point on the time-line (namely, the start time of
4.2 Scheduling to Minimize Lateness: An Exchange Argument

Ij), and so \( t + 1 \leq d \). Thus \( t \leq d - 1 \). It follows that at least one of the \( d \) labels is not excluded by this set of \( t \) intervals, and so there is a label that can be assigned to \( I_j \).

Next we claim that no two overlapping intervals are assigned the same label. Indeed, consider any two intervals \( I \) and \( I' \) that overlap, and suppose \( I \) precedes \( I' \) in the sorted order. Then when \( I' \) is considered by the algorithm, \( I \) is in the set of intervals whose labels are excluded from consideration; consequently, the algorithm will not assign to \( I' \) the label that it used for \( I \).

The algorithm and its analysis are very simple. Essentially, if you have \( d \) labels at your disposal, then as you sweep through the intervals from left to right, assigning an available label to each interval you encounter, you can never reach a point where all the labels are currently in use.

Since our algorithm is using \( d \) labels, we can use (4.4) to conclude that it is, in fact, always using the minimum possible number of labels. We sum this up as follows.

\[
(4.6) \quad \text{The greedy algorithm above schedules every interval on a resource, using a number of resources equal to the depth of the set of intervals. This is the optimal number of resources needed.}
\]

4.2 Scheduling to Minimize Lateness: An Exchange Argument

We now discuss a scheduling problem related to the one with which we began the chapter. Despite the similarities in the problem formulation and in the greedy algorithm to solve it, the proof that this algorithm is optimal will require a more sophisticated kind of analysis.

The Problem

Consider again a situation in which we have a single resource and a set of \( n \) requests to use the resource for an interval of time. Assume that the resource is available starting at time \( s \). In contrast to the previous problem, however, each request is now more flexible. Instead of a start time and finish time, the request \( i \) has a deadline \( d_i \), and it requires a contiguous time interval of length \( t_i \), but it is willing to be scheduled at any time before the deadline. Each accepted request must be assigned an interval of time of length \( t_i \), and different requests must be assigned nonoverlapping intervals.

There are many objective functions we might seek to optimize when faced with this situation, and some are computationally much more difficult than
Chapter 4  Greedy Algorithms

Figure 4.5 A sample instance of scheduling to minimize lateness.

others. Here we consider a very natural goal that can be optimized by a greedy algorithm. Suppose that we plan to satisfy each request, but we are allowed to let certain requests run late. Thus, beginning at our overall start time \( s \), we will assign each request \( i \) an interval of time of length \( t_i \); let us denote this interval by \([s(i), f(i)]\), with \( f(i) = s(i) + t_i \). Unlike the previous problem, then, the algorithm must actually determine a start time (and hence a finish time) for each interval.

We say that a request \( i \) is late if it misses the deadline, that is, if \( f(i) > d_i \). The lateness of such a request \( i \) is defined to be \( l_i = f(i) - d_i \). We will say that \( l_i = 0 \) if request \( i \) is not late. The goal in our new optimization problem will be to schedule all requests, using nonoverlapping intervals, so as to minimize the maximum lateness, \( L = \max_i l_i \). This problem arises naturally when scheduling jobs that need to use a single machine, and so we will refer to our requests as jobs.

Figure 4.5 shows a sample instance of this problem, consisting of three jobs: the first has length \( t_1 = 1 \) and deadline \( d_1 = 2 \); the second has \( t_2 = 2 \) and \( d_2 = 4 \); and the third has \( t_3 = 3 \) and \( d_3 = 6 \). It is not hard to check that scheduling the jobs in the order 1, 2, 3 incurs a maximum lateness of 0.

Designing the Algorithm

What would a greedy algorithm for this problem look like? There are several natural greedy approaches in which we look at the data \((t_i, d_i)\) about the jobs and use this to order them according to some simple rule.

- One approach would be to schedule the jobs in order of increasing length \( t_i \), so as to get the short jobs out of the way quickly. This immediately
looks too simplistic, since it completely ignores the deadlines of the jobs. And indeed, consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 100$, while the second job has $t_2 = 10$ and $d_2 = 10$. Then the second job has to be started right away if we want to achieve lateness $L = 0$, and scheduling the second job first is indeed the optimal solution.

The previous example suggests that we should be concerned about jobs whose available *slack time* $d_i - t_i$ is very small—they’re the ones that need to be started with minimal delay. So a more natural greedy algorithm would be to sort jobs in order of increasing slack $d_i - t_i$.

Unfortunately, this greedy rule fails as well. Consider a two-job instance where the first job has $t_1 = 1$ and $d_1 = 2$, while the second job has $t_2 = 10$ and $d_2 = 10$. Sorting by increasing slack would place the second job first in the schedule, and the first job would incur a lateness of 9. (It finishes at time 11, nine units beyond its deadline.) On the other hand, if we schedule the first job first, then it finishes on time and the second job incurs a lateness of only 1.

There is, however, an equally basic greedy algorithm that always produces an optimal solution. We simply sort the jobs in increasing order of their deadlines $d_i$, and schedule them in this order. (This rule is often called *Earliest Deadline First*.) There is an intuitive basis to this rule: we should make sure that jobs with earlier deadlines get completed earlier. At the same time, it’s a little hard to believe that this algorithm always produces optimal solutions—specifically because it never looks at the lengths of the jobs. Earlier we were skeptical of the approach that sorted by length on the grounds that it threw away half the input data (i.e., the deadlines); but now we’re considering a solution that throws away the other half of the data. Nevertheless, Earliest Deadline First does produce optimal solutions, and we will now prove this.

First we specify some notation that will be useful in talking about the algorithm. By renaming the jobs if necessary, we can assume that the jobs are labeled in the order of their deadlines, that is, we have

$$d_1 \leq \ldots \leq d_n.$$  

We will simply schedule all jobs in this order. Again, let $s$ be the start time for all jobs. Job 1 will start at time $s = s(1)$ and end at time $f(1) = s(1) + t_1$; Job 2 will start at time $s(2) = f(1)$ and end at time $f(2) = s(2) + t_2$; and so forth. We will use $f$ to denote the finishing time of the last scheduled job. We write this algorithm here.

---

Order the jobs in order of their deadlines
Assume for simplicity of notation that $d_1 \leq \ldots \leq d_n$
Initially, $f = s$
Consider the jobs \( i = 1, \ldots, n \) in this order

Assign job \( i \) to the time interval from \( s(i) = f \) to \( f(i) = f + t_i \)

Let \( f = f + t_i \)
End

Return the set of scheduled intervals \([s(i), f(i)]\) for \( i = 1, \ldots, n\)

---

**Analyzing the Algorithm**

To reason about the optimality of the algorithm, we first observe that the schedule it produces has no “gaps”—times when the machine is not working yet there are jobs left. The time that passes during a gap will be called idle time: there is work to be done, yet for some reason the machine is sitting idle. Not only does the schedule \( A \) produced by our algorithm have no idle time; it is also very easy to see that there is an optimal schedule with this property. We do not write down a proof for this.

(4.7) **There is an optimal schedule with no idle time.**

Now, how can we prove that our schedule \( A \) is optimal, that is, its maximum lateness \( L \) is as small as possible? As in previous analyses, we will start by considering an optimal schedule \( O \). Our plan here is to gradually modify \( O \), preserving its optimality at each step, but eventually transforming it into a schedule that is identical to the schedule \( A \) found by the greedy algorithm. We refer to this type of analysis as an exchange argument, and we will see that it is a powerful way to think about greedy algorithms in general.

We first try characterizing schedules in the following way. We say that a schedule \( A' \) has an inversion if a job \( i \) with deadline \( d_i \) is scheduled before another job \( j \) with earlier deadline \( d_j < d_i \). Notice that, by definition, the schedule \( A \) produced by our algorithm has no inversions. If there are jobs with identical deadlines then there can be many different schedules with no inversions. However, we can show that all these schedules have the same maximum lateness \( L \).

(4.8) **All schedules with no inversions and no idle time have the same maximum lateness.**

**Proof.** If two different schedules have neither inversions nor idle time, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. Consider such a deadline \( d \). In both schedules, the jobs with deadline \( d \) are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines). Among the jobs with deadline \( d \), the last one has the greatest lateness, and this lateness does not depend on the order of the jobs. □
4.2 Scheduling to Minimize Lateness: An Exchange Argument

The main step in showing the optimality of our algorithm is to establish that there is an optimal schedule that has no inversions and no idle time. To do this, we will start with any optimal schedule having no idle time; we will then convert it into a schedule with no inversions without increasing its maximum lateness. Thus the resulting scheduling after this conversion will be optimal as well.

(4.9) There is an optimal schedule that has no inversions and no idle time.

Proof. By (4.7), there is an optimal schedule \( O \) with no idle time. The proof will consist of a sequence of statements. The first of these is simple to establish.

(a) If \( O \) has an inversion, then there is a pair of jobs \( i \) and \( j \) such that \( j \) is scheduled immediately after \( i \) and has \( d_j < d_i \).

Indeed, consider an inversion in which a job \( a \) is scheduled sometime before a job \( b \), and \( d_a > d_b \). If we advance in the scheduled order of jobs from \( a \) to \( b \) one at a time, there has to come a point at which the deadline we see decreases for the first time. This corresponds to a pair of consecutive jobs that form an inversion.

Now suppose \( O \) has at least one inversion, and by (a), let \( i \) and \( j \) be a pair of inverted requests that are consecutive in the scheduled order. We will decrease the number of inversions in \( O \) by swapping the requests \( i \) and \( j \) in the schedule \( O \). The pair \((i, j)\) formed an inversion in \( O \), this inversion is eliminated by the swap, and no new inversions are created. Thus we have

(b) After swapping \( i \) and \( j \) we get a schedule with one less inversion.

The hardest part of this proof is to argue that the inverted schedule is also optimal.

(c) The new swapped schedule has a maximum lateness no larger than that of \( O \).

It is clear that if we can prove (c), then we are done. The initial schedule \( O \) can have at most \( \binom{n}{2} \) inversions (if all pairs are inverted), and hence after at most \( \binom{n}{2} \) swaps we get an optimal schedule with no inversions.

So we now conclude by proving (c), showing that by swapping a pair of consecutive, inverted jobs, we do not increase the maximum lateness \( L \) of the schedule.

Proof of (c). We invent some notation to describe the schedule \( O \): assume that each request \( r \) is scheduled for the time interval \([s(r), f(r)]\) and has lateness \( l'_r \). Let \( L' = \max_r l'_r \) denote the maximumlateness of this schedule.

For any query, contact us at help@tutorialsduniya.com
Chapter 4  Greedy Algorithms

Let $O$ denote the swapped schedule; we will use $s(r)$, $f(r)$, $L_r$, and $L$ to denote the corresponding quantities in the swapped schedule.

Now recall our two adjacent, inverted jobs $i$ and $j$. The situation is roughly as pictured in Figure 4.6. The finishing time of $j$ before the swap is exactly equal to the finishing time of $i$ after the swap. Thus all jobs other than jobs $i$ and $j$ finish at the same time in the two schedules. Moreover, job $j$ will get finished earlier in the new schedule, and hence the swap does not increase the lateness of job $j$.

Thus the only thing to worry about is job $i$: its lateness may have been increased, and what if this actually raises the maximum lateness of the whole schedule? After the swap, job $i$ finishes at time $f(j)$, when job $j$ was finished in the schedule $O$. If job $i$ is late in this new schedule, its lateness is $\bar{L}_i = f(i) - d_i = f(j) - d_i$. But the crucial point is that $i$ cannot be more late in the schedule $O$ than $j$ was in the schedule $O$. Specifically, our assumption $d_i > d_j$ implies that

$$\bar{L}_i = f(j) - d_i < f(j) - d_j = \bar{L}_j,$$

Since the lateness of the schedule $O$ was $L' \geq \bar{L}_j > \bar{L}_i$, this shows that the swap does not increase the maximum lateness of the schedule. ■

The optimality of our greedy algorithm now follows immediately.
4.3 Optimal Caching: A More Complex Exchange Argument

(4.10) The schedule A produced by the greedy algorithm has optimal maximum lateness L.

Proof. Statement (4.9) proves that an optimal schedule with no inversions exists. Now by (4.8) all schedules with no inversions have the same maximum lateness, and so the schedule obtained by the greedy algorithm is optimal. ■

Extensions

There are many possible generalizations of this scheduling problem. For example, we assumed that all jobs were available to start at the common start time s. A natural, but harder, version of this problem would contain requests i that, in addition to the deadline $d_i$ and the requested time $t_i$, would also have an earliest possible starting time $r_i$. This earliest possible starting time is usually referred to as the release time. Problems with release times arise naturally in scheduling problems where requests can take the form: Can I reserve the room for a two-hour lecture, sometime between 1 P.M. and 5 P.M.? Our proof that the greedy algorithm finds an optimal solution relied crucially on the fact that all jobs were available at the common start time s. (Do you see where?) Unfortunately, as we will see later in the book, in Chapter 8, this more general version of the problem is much more difficult to solve optimally.
4.4 Shortest Paths in a Graph

Some of the basic algorithms for graphs are based on greedy design principles. Here we apply a greedy algorithm to the problem of finding shortest paths, and in the next section we look at the construction of minimum-cost spanning trees.

The Problem

As we’ve seen, graphs are often used to model networks in which one travels from one point to another—traversing a sequence of highways through interchanges, or traversing a sequence of communication links through intermediate routers. As a result, a basic algorithmic problem is to determine the shortest path between nodes in a graph. We may ask this as a point-to-point question: Given nodes $u$ and $v$, what is the shortest $u$-$v$ path? Or we may ask for more information: Given a start node $s$, what is the shortest path from $s$ to each other node?

The concrete setup of the shortest paths problem is as follows. We are given a directed graph $G = (V, E)$, with a designated start node $s$. We assume that $s$ has a path to every other node in $G$. Each edge $e$ has a length $\ell_e \geq 0$, indicating the time (or distance, or cost) it takes to traverse $e$. For a path $P$, the length of $P$—denoted $\ell(P)$—is the sum of the lengths of all edges in $P$. Our goal is to determine the shortest path from $s$ to every other node in the graph. We should mention that although the problem is specified for a directed graph, we can handle the case of an undirected graph by simply replacing each undirected edge $e = (u, v)$ of length $\ell_e$ by two directed edges $(u, v)$ and $(v, u)$, each of length $\ell_e$.

Designing the Algorithm

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest-paths problem. We begin by describing an algorithm that just determines the length of the shortest path from $s$ to each other node in the graph; it is then easy to produce the paths as well. The algorithm maintains a set $S$ of vertices $u$ for which we have determined a shortest-path distance $d(u)$ from $s$; this is the “explored” part of the graph. Initially $S = \{s\}$, and $d(s) = 0$. Now, for each node $v \in V - S$, we determine the shortest path that can be constructed by traveling along a path through the explored part $S$ to some $u \in S$, followed by the single edge $(u, v)$. That is, we consider the quantity
Chapter 4  Greedy Algorithms

\[ d'(v) = \min_{e=(u,v)\in S} d(u) + \ell_e. \]
We choose the node \( v \in V - S \) for which this quantity is minimized, add \( v \) to \( S \), and define \( d(v) \) to be the value \( d'(v) \).

---

**Dijkstra's Algorithm** \((G, \ell)\)

Let \( S \) be the set of explored nodes

For each \( u \in S \), we store a distance \( d(u) \)

Initially \( S = \{s\} \) and \( d(s) = 0 \)

While \( S \neq V \)

Select a node \( v \notin S \) with at least one edge from \( S \) for which \( d'(v) = \min_{e=(u,v)\in S} d(u) + \ell_e \) is as small as possible

Add \( v \) to \( S \) and define \( d(v) = d'(v) \)

EndWhile

---

It is simple to produce the \( s-u \) paths corresponding to the distances found by Dijkstra's Algorithm. As each node \( v \) is added to the set \( S \), we simply record the edge \((u, v)\) on which it achieved the value \( \min_{e=(u,v)\in S} d(u) + \ell_e \). The path \( P_v \) is implicitly represented by these edges: if \((u, v)\) is the edge we have stored for \( v \), then \( P_v \) is just (recursively) the path \( P_u \) followed by the single edge \((u, v)\). In other words, to construct \( P_v \), we simply start at \( v \); follow the edge we have stored for \( v \) in the reverse direction to \( u \); then follow the edge we have stored for \( u \) in the reverse direction to its predecessor; and so on until we reach \( s \). Note that \( s \) must be reached, since our backward walk from \( v \) visits nodes that were added to \( S \) earlier and earlier.

To get a better sense of what the algorithm is doing, consider the snapshot of its execution depicted in Figure 4.7. At the point the picture is drawn, two iterations have been performed: the first added node \( u \), and the second added node \( v \). In the iteration that is about to be performed, the node \( x \) will be added because it achieves the smallest value of \( d'(x) \); thanks to the edge \((u, x)\), we have \( d'(x) = d(u) + \ell_{ux} = 2 \). Note that attempting to add \( y \) or \( z \) to the set \( S \) at this point would lead to an incorrect value for their shortest-path distances; ultimately, they will be added because of their edges from \( x \).

---

**Analyzing the Algorithm**

We see in this example that Dijkstra’s Algorithm is doing the right thing and avoiding recurring pitfalls: growing the set \( S \) by the wrong node can lead to an overestimate of the shortest-path distance to that node. The question becomes: Is it always true that when Dijkstra’s Algorithm adds a node \( v \), we get the true shortest-path distance to \( v \)?

We now answer this by proving the correctness of the algorithm, showing that the paths \( P_u \) really are shortest paths. Dijkstra’s Algorithm is greedy in
4.4 Shortest Paths in a Graph

Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set $S$ is $x$, due to the path through $u$.

the sense that we always form the shortest new $s$-$v$ path we can make from a path in $S$ followed by a single edge. We prove its correctness using a variant of our first style of analysis: we show that it “stays ahead” of all other solutions by establishing, inductively, that each time it selects a path to a node $v$, that path is shorter than every other possible path to $v$.

(4.14) Consider the set $S$ at any point in the algorithm’s execution. For each $u \in S$, the path $P_u$ is a shortest $s$-$u$ path.

Note that this fact immediately establishes the correctness of Dijkstra’s Algorithm, since we can apply it when the algorithm terminates, at which point $S$ includes all nodes.

Proof. We prove this by induction on the size of $S$. The case $|S| = 1$ is easy, since then we have $S = \{s\}$ and $d(s) = 0$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; we now grow $S$ to size $k + 1$ by adding the node $v$. Let $(u, v)$ be the final edge on our $s$-$v$ path $P_v$.

By induction hypothesis, $P_u$ is the shortest $s$-$u$ path for each $u \in S$. Now consider any other $s$-$v$ path $P$; we wish to show that it is at least as long as $P_v$. In order to reach $v$, this path $P$ must leave the set $S$ somewhere; let $y$ be the first node on $P$ that is not in $S$, and let $x \in S$ be the node just before $y$.

The situation is now as depicted in Figure 4.8, and the crux of the proof is very simple: $P$ cannot be shorter than $P_v$, because it is already at least as
Chapter 4  Greedy Algorithms

long as \( P_v \) by the time it has left the set \( S \). Indeed, in iteration \( k + 1 \), Dijkstra’s Algorithm must have considered adding node \( y \) to the set \( S \) via the edge \((x, y)\) and rejected this option in favor of adding \( v \). This means that there is no path from \( s \) to \( y \) through \( x \) that is shorter than \( P_v \). But the subpath of \( P \) up to \( y \) is such a path, and so this subpath is at least as long as \( P_v \). Since edge lengths are nonnegative, the full path \( P \) is at least as long as \( P_v \) as well.

This is a complete proof; one can also spell out the argument in the previous paragraph using the following inequalities. Let \( P' \) be the subpath of \( P \) from \( s \) to \( x \). Since \( x \in S \), we know by the induction hypothesis that \( P_x \) is a shortest \( s-x \) path (of length \( d(x) \)), and so \( \ell(P') \geq \ell(P_x) = d(x) \). Thus the subpath of \( P \) out to node \( y \) has length \( \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq d'(y) \), and the full path \( P \) is at least as long as this subpath. Finally, since Dijkstra’s Algorithm selected \( v \) in this iteration, we know that \( d'(y) \geq d'(v) = \ell(P_v) \). Combining these inequalities shows that \( \ell(P) \geq \ell(P') + \ell(x, y) \geq \ell(P_v) \). □

Here are two observations about Dijkstra’s Algorithm and its analysis. First, the algorithm does not always find shortest paths if some of the edges can have negative lengths. (Do you see where the proof breaks?) Many shortest-path applications involve negative edge lengths, and a more complex algorithm—due to Bellman and Ford—is required for this case. We will see this algorithm when we consider the topic of dynamic programming.

The second observation is that Dijkstra’s Algorithm is, in a sense, even simpler than we’ve described here. Dijkstra’s Algorithm is really a “continuous” version of the standard breadth-first search algorithm for traversing a graph, and it can be motivated by the following physical intuition. Suppose the edges of \( G \) formed a system of pipes filled with water, joined together at the nodes; each edge \( e \) has length \( \ell_e \) and a fixed cross-sectional area. Now suppose an extra droplet of water falls at node \( s \) and starts a wave from \( s \). As the wave expands out of node \( s \) at a constant speed, the expanding sphere
4.4 Shortest Paths in a Graph

of wavefront reaches nodes in increasing order of their distance from \( s \). It is easy to believe (and also true) that the path taken by the wavefront to get to any node \( v \) is a shortest path. Indeed, it is easy to see that this is exactly the path to \( v \) found by Dijkstra’s Algorithm, and that the nodes are discovered by the expanding water in the same order that they are discovered by Dijkstra’s Algorithm.

**Implementation and Running Time** To conclude our discussion of Dijkstra’s Algorithm, we consider its running time. There are \( n - 1 \) iterations of the While loop for a graph with \( n \) nodes, as each iteration adds a new node \( v \) to \( S \). Selecting the correct node \( v \) efficiently is a more subtle issue. One’s first impression is that each iteration would have to consider each node \( v \not\in S \), and go through all the edges between \( S \) and \( v \) to determine the minimum

\[
\min_{e=(u,v):u \in S} d(u) + \ell_e,
\]

so that we can select the node \( v \) for which this minimum is smallest. For a graph with \( m \) edges, computing all these minima can take \( O(m) \) time, so this would lead to an implementation that runs in \( O(mn) \) time.

We can do considerably better if we use the right data structures. First, we will explicitly maintain the values of the minima \( d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e \) for each node \( v \in V - S \), rather than recomputing them in each iteration. We can further improve the efficiency by keeping the nodes \( V - S \) in a priority queue with \( d'(v) \) as their keys. Priority queues were discussed in Chapter 2; they are data structures designed to maintain a set of \( n \) elements, each with a key. A priority queue can efficiently insert elements, delete elements, change an element’s key, and extract the element with the minimum key. We will need the third and fourth of the above operations: ChangeKey and ExtractMin.

How do we implement Dijkstra’s Algorithm using a priority queue? We put the nodes \( V \) in a priority queue with \( d'(v) \) as the key for \( v \in V \). To select the node \( v \) that should be added to the set \( S \), we need the ExtractMin operation. To see how to update the keys, consider an iteration in which node \( v \) is added to \( S \), and let \( w \not\in S \) be a node that remains in the priority queue. What do we have to do to update the value of \( d'(w) \)? If \((v, w)\) is not an edge, then we don’t have to do anything; the set of edges considered in the minimum

\[
\min_{e=(u,w):u \in S} d(u) + \ell_e
\]

is exactly the same before and after adding \( v \) to \( S \). If \( e' = (v, w) \in E \), on the other hand, then the new value for the key is \( \min(d'(w), d(v) + \ell_{e'}) \). If \( d'(w) > d(v) + \ell_{e'} \) then we need to use the ChangeKey operation to decrease the key of node \( w \) appropriately. This ChangeKey operation can occur at most once per edge, when the tail of the edge \( e' \) is added to \( S \). In summary, we have the following result.
(4.15) Using a priority queue, Dijkstra’s Algorithm can be implemented on a graph with \( n \) nodes and \( m \) edges to run in \( O(m) \) time, plus the time for \( n \) ExtractMin and \( m \) ChangeKey operations.

Using the heap-based priority queue implementation discussed in Chapter 2, each priority queue operation can be made to run in \( O(\log n) \) time. Thus the overall time for the implementation is \( O(m \log n) \).

4.5 The Minimum Spanning Tree Problem

We now apply an exchange argument in the context of a second fundamental problem on graphs: the Minimum Spanning Tree Problem.

**The Problem**

Suppose we have a set of locations \( V = \{v_1, v_2, \ldots, v_n\} \), and we want to build a communication network on top of them. The network should be connected—there should be a path between every pair of nodes—but subject to this requirement, we wish to build it as cheaply as possible.

For certain pairs \((v_i, v_j)\), we may build a direct link between \( v_i \) and \( v_j \) for a certain cost \( c(v_i, v_j) > 0 \). Thus we can represent the set of possible links that may be built using a graph \( G = (V, E) \), with a positive cost \( c_e \) associated with each edge \( e = (v_i, v_j) \). The problem is to find a subset of the edges \( T \subseteq E \) so that the graph \( (V, T) \) is connected, and the total cost \( \sum_{e \in T} c_e \) is as small as possible. (We will assume that the full graph \( G \) is connected; otherwise, no solution is possible.)

Here is a basic observation.

(4.16) Let \( T \) be a minimum-cost solution to the network design problem defined above. Then \( (V, T) \) is a tree.

**Proof.** By definition, \( (V, T) \) must be connected; we show that it also will contain no cycles. Indeed, suppose it contained a cycle \( C \), and let \( e \) be any edge on \( C \). We claim that \( (V, T - \{e\}) \) is still connected, since any path that previously used the edge \( e \) can now go “the long way” around the remainder of the cycle \( C \) instead. It follows that \( (V, T - \{e\}) \) is also a valid solution to the problem, and it is cheaper—a contradiction. \( \blacksquare \)

If we allow some edges to have 0 cost (that is, we assume only that the costs \( c_e \) are nonnegative), then a minimum-cost solution to the network design problem may have extra edges—edges that have 0 cost and could optionally be deleted. But even in this case, there is always a minimum-cost solution that is a tree. Starting from any optimal solution, we could keep deleting edges on
cycles until we had a tree; with nonnegative edges, the cost would not increase during this process.

We will call a subset $T \subseteq E$ a spanning tree of $G$ if $(V, T)$ is a tree. Statement (4.16) says that the goal of our network design problem can be rephrased as that of finding the cheapest spanning tree of the graph; for this reason, it is generally called the Minimum Spanning Tree Problem. Unless $G$ is a very simple graph, it will have exponentially many different spanning trees, whose structures may look very different from one another. So it is not at all clear how to efficiently find the cheapest tree from among all these options.

Designing Algorithms
As with the previous problems we’ve seen, it is easy to come up with a number of natural greedy algorithms for the problem. But curiously, and fortunately, this is a case where many of the first greedy algorithms one tries turn out to be correct: they each solve the problem optimally. We will review a few of these algorithms now and then discover, via a nice pair of exchange arguments, some of the underlying reasons for this plethora of simple, optimal algorithms.

Here are three greedy algorithms, each of which correctly finds a minimum spanning tree.

- One simple algorithm starts without any edges at all and builds a spanning tree by successively inserting edges from $E$ in order of increasing cost. As we move through the edges in this order, we insert each edge $e$ as long as it does not create a cycle when added to the edges we’ve already inserted. If, on the other hand, inserting $e$ would result in a cycle, then we simply discard $e$ and continue. This approach is called Kruskal’s Algorithm.

- Another simple greedy algorithm can be designed by analogy with Dijkstra’s Algorithm for paths, although, in fact, it is even simpler to specify than Dijkstra’s Algorithm. We start with a root node $s$ and try to greedily grow a tree from $s$ outward. At each step, we simply add the node that can be attached as cheaply as possibly to the partial tree we already have. More concretely, we maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far. Initially, $S = \{s\}$. In each iteration, we grow $S$ by one node, adding the node $v$ that minimizes the “attachment cost” $\min_{e=(u,v): u \in S} C_e$, and including the edge $e = (u, v)$ that achieves this minimum in the spanning tree. This approach is called Prim’s Algorithm.

- Finally, we can design a greedy algorithm by running sort of a “backward” version of Kruskal’s Algorithm. Specifically, we start with the full graph $(V, E)$ and begin deleting edges in order of decreasing cost. As we get to each edge $e$ (starting from the most expensive), we delete it as
long as doing so would not actually disconnect the graph we currently have. For want of a better name, this approach is generally called the Reverse-Delete Algorithm (as far as we can tell, it’s never been named after a specific person).

For example, Figure 4.9 shows the first four edges added by Prim’s and Kruskal’s Algorithms respectively, on a geometric instance of the Minimum Spanning Tree Problem in which the cost of each edge is proportional to the geometric distance in the plane.

The fact that each of these algorithms is guaranteed to produce an optimal solution suggests a certain “robustness” to the Minimum Spanning Tree Problem—there are many ways to get to the answer. Next we explore some of the underlying reasons why so many different algorithms produce minimum-cost spanning trees.

Analyzing the Algorithms

All these algorithms work by repeatedly inserting or deleting edges from a partial solution. So, to analyze them, it would be useful to have in hand some basic facts saying when it is “safe” to include an edge in the minimum spanning tree, and, correspondingly, when it is safe to eliminate an edge on the grounds that it couldn’t possibly be in the minimum spanning tree. For purposes of the analysis, we will make the simplifying assumption that all edge costs are distinct from one another (i.e., no two are equal). This assumption makes it
4.5 The Minimum Spanning Tree Problem

easier to express the arguments that follow, and we will show later in this section how this assumption can be easily eliminated.

**When Is It Safe to Include an Edge in the Minimum Spanning Tree?** The crucial fact about edge insertion is the following statement, which we will refer to as the Cut Property.

\[(4.17)\] Assume that all edge costs are distinct. Let $S$ be any subset of nodes that is neither empty nor equal to all of $V$, and let edge $e = (v, w)$ be the minimum-cost edge with one end in $S$ and the other in $V - S$. Then every minimum spanning tree contains the edge $e$.

**Proof.** Let $T$ be a spanning tree that does not contain $e$; we need to show that $T$ does not have the minimum possible cost. We’ll do this using an exchange argument: we’ll identify an edge $e'$ in $T$ that is more expensive than $e$, and with the property exchanging $e$ for $e'$ results in another spanning tree. This resulting spanning tree will then be cheaper than $T$, as desired.

The crux is therefore to find an edge that can be successfully exchanged with $e$. Recall that the ends of $e$ are $v$ and $w$. $T$ is a spanning tree, so there must be a path $P$ in $T$ from $v$ to $w$. Starting at $v$, suppose we follow the nodes of $P$ in sequence; there is a first node $w'$ on $P$ that is in $V - S$. Let $v' \in S$ be the node just before $w'$ on $P$, and let $e' = (v', w')$ be the edge joining them. Thus, $e'$ is an edge of $T$ with one end in $S$ and the other in $V - S$. See Figure 4.10 for the situation at this stage in the proof.

If we exchange $e$ for $e'$, we get a set of edges $T' = T - \{e'\} \cup \{e\}$. We claim that $T'$ is a spanning tree. Clearly $(V, T')$ is connected, since $(V, T)$ is connected, and any path in $(V, T)$ that used the edge $e' = (v', w')$ can now be “rerouted” in $(V, T')$ to follow the portion of $P$ from $v'$ to $v$, then the edge $e$, and then the portion of $P$ from $w$ to $w'$. To see that $(V, T')$ is also acyclic, note that the only cycle in $(V, T' \cup \{e'\})$ is the one composed of $e$ and the path $P$, and this cycle is not present in $(V, T')$ due to the deletion of $e'$.

We noted above that the edge $e'$ has one end in $S$ and the other in $V - S$. But $e$ is the cheapest edge with this property, and so $c_e < c_{e'}$. (The inequality is strict since no two edges have the same cost.) Thus the total cost of $T'$ is less than that of $T$, as desired. ■

The proof of (4.17) is a bit more subtle than it may first appear. To appreciate this subtlety, consider the following shorter but incorrect argument for (4.17). Let $T$ be a spanning tree that does not contain $e$. Since $T$ is a spanning tree, it must contain an edge $f$ with one end in $S$ and the other in $V - S$. Since $e$ is the cheapest edge with this property, we have $c_e < c_f$, and hence $T - \{f\} \cup \{e\}$ is a spanning tree that is cheaper than $T$.

For any query, contact us at help@todaysduniya.com
Chapter 4  Greedy Algorithms

The problem with this argument is not in the claim that \( f \) exists, or that \( T - \{ f \} \cup \{ e \} \) is cheaper than \( T \). The difficulty is that \( T - \{ f \} \cup \{ e \} \) may not be a spanning tree, as shown by the example of the edge \( f \) in Figure 4.10. The point is that we can’t prove (4.17) by simply picking any edge in \( T \) that crosses from \( S \) to \( V - S \); some care must be taken to find the right one.

**The Optimality of Kruskal’s and Prim’s Algorithms**  We can now easily prove the optimality of both Kruskal’s Algorithm and Prim’s Algorithm. The point is that both algorithms only include an edge when it is justified by the Cut Property (4.17).

\[ (4.18) \text{Kruskal’s Algorithm produces a minimum spanning tree of } G. \]

**Proof.** Consider any edge \( e = (v, w) \) added by Kruskal’s Algorithm, and let \( S \) be the set of all nodes to which \( v \) has a path at the moment just before \( e \) is added. Clearly \( v \in S \), but \( w \notin S \), since adding \( e \) does not create a cycle. Moreover, no edge from \( S \) to \( V - S \) has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have been added by Kruskal’s Algorithm. Thus \( e \) is the cheapest edge with one end in \( S \) and the other in \( V - S \), and so by (4.17) it belongs to every minimum spanning tree.
4.5 The Minimum Spanning Tree Problem

So if we can show that the output \((V, T)\) of Kruskal’s Algorithm is in fact a spanning tree of \(G\), then we will be done. Clearly \((V, T)\) contains no cycles, since the algorithm is explicitly designed to avoid creating cycles. Further, if \((V, T)\) were not connected, then there would exist a nonempty subset of nodes \(S\) (not equal to all of \(V\)) such that there is no edge from \(S\) to \(V - S\). But this contradicts the behavior of the algorithm: we know that since \(G\) is connected, there is at least one edge between \(S\) and \(V - S\), and the algorithm will add the first of these that it encounters.

\[
\text{(4.19) Prim’s Algorithm produces a minimum spanning tree of } G.
\]

**Proof.** For Prim’s Algorithm, it is also very easy to show that it only adds edges belonging to every minimum spanning tree. Indeed, in each iteration of the algorithm, there is a set \(S \subseteq V\) on which a partial spanning tree has been constructed, and a node \(v\) and edge \(e\) are added that minimize the quantity \(\min_{e = (u, v) : u \in S} c_e\). By definition, \(e\) is the cheapest edge with one end in \(S\) and the other end in \(V - S\), and so by the Cut Property (4.17) it is in every minimum spanning tree.

It is also straightforward to show that Prim’s Algorithm produces a spanning tree of \(G\), and hence it produces a minimum spanning tree.

**When Can We Guarantee an Edge Is Not in the Minimum Spanning Tree?** The crucial fact about edge deletion is the following statement, which we will refer to as the *Cycle Property*.

\[
\text{(4.20) Assume that all edge costs are distinct. Let } C \text{ be any cycle in } G, \text{ and let edge } e = (v, w) \text{ be the most expensive edge belonging to } C. \text{ Then } e \text{ does not belong to any minimum spanning tree of } G.
\]

**Proof.** Let \(T\) be a spanning tree that contains \(e\); we need to show that \(T\) does not have the minimum possible cost. By analogy with the proof of the Cut Property (4.17), we’ll do this with an exchange argument, swapping \(e\) for a cheaper edge in such a way that we still have a spanning tree.

So again the question is: How do we find a cheaper edge that can be exchanged in this way with \(e\)? Let’s begin by deleting \(e\) from \(T\); this partitions the nodes into two components: \(S\), containing node \(v\); and \(V - S\), containing node \(w\). Now, the edge we use in place of \(e\) should have one end in \(S\) and the other in \(V - S\), so as to stitch the tree back together.

We can find such an edge by following the cycle \(C\). The edges of \(C\) other than \(e\) form, by definition, a path \(P\) with one end at \(v\) and the other at \(w\). If we follow \(P\) from \(v\) to \(w\), we begin in \(S\) and end up in \(V - S\), so there is some
Figure 4.11 Swapping the edge \( e' \) for the edge \( e \) in the spanning tree \( T \), as described in the proof of (4.20).

edge \( e' \) on \( P \) that crosses from \( S \) to \( V - S \). See Figure 4.11 for an illustration of this.

Now consider the set of edges \( T' = T - \{e\} \cup \{e'\} \). Arguing just as in the proof of the Cut Property (4.17), the graph \( (V, T') \) is connected and has no cycles, so \( T' \) is a spanning tree of \( G \). Moreover, since \( e \) is the most expensive edge on the cycle \( C \), and \( e' \) belongs to \( C \), it must be that \( e' \) is cheaper than \( e \), and hence \( T' \) is cheaper than \( T \), as desired.
4.6 Implementing Kruskal’s Algorithm: The Union-Find Data Structure

One of the most basic graph problems is to find the set of connected components. In Chapter 3 we discussed linear-time algorithms using BFS or DFS for finding the connected components of a graph.

In this section, we consider the scenario in which a graph evolves through the addition of edges. That is, the graph has a fixed population of nodes, but it grows over time by having edges appear between certain pairs of nodes. Our goal is to maintain the set of connected components of such a graph throughout this evolution process. When an edge is added to the graph, we don’t want to have to recompute the connected components from scratch. Rather, we will develop a data structure that we call the Union-Find structure, which will store a representation of the components in a way that supports rapid searching and updating.

This is exactly the data structure needed to implement Kruskal’s Algorithm efficiently. As each edge \( e = (v, w) \) is considered, we need to efficiently find the identities of the connected components containing \( v \) and \( w \). If these components are different, then there is no path from \( v \) and \( w \), and hence edge \( e \) should be included; but if the components are the same, then there is a \( v-w \) path on the edges already included, and so \( e \) should be omitted. In the event that \( e \) is included, the data structure should also support the efficient merging of the components of \( v \) and \( w \) into a single new component.

The Problem

The Union-Find data structure allows us to maintain disjoint sets (such as the components of a graph) in the following sense. Given a node \( u \), the operation \( \text{Find}(u) \) will return the name of the set containing \( u \). This operation can be used to test if two nodes \( u \) and \( v \) are in the same set, by simply checking if \( \text{Find}(u) = \text{Find}(v) \). The data structure will also implement an operation \( \text{Union}(A, B) \) to take two sets \( A \) and \( B \) and merge them to a single set.

These operations can be used to maintain connected components of an evolving graph \( G = (V, E) \) as edges are added. The sets will be the connected components of the graph. For a node \( u \), the operation \( \text{Find}(u) \) will return the
name of the component containing $u$. If we add an edge $(u, v)$ to the graph, then we first test if $u$ and $v$ are already in the same connected component (by testing if $\text{Find}(u) = \text{Find}(v)$). If they are not, then $\text{Union}(\text{Find}(u), \text{Find}(v))$ can be used to merge the two components into one. It is important to note that the Union-Find data structure can only be used to maintain components of a graph as we add edges; it is not designed to handle the effects of edge deletion, which may result in a single component being “split” into two.

To summarize, the Union-Find data structure will support three operations.

- $\text{MakeUnionFind}(S)$ for a set $S$ will return a Union-Find data structure on set $S$ where all elements are in separate sets. This corresponds, for example, to the connected components of a graph with no edges. Our goal will be to implement $\text{MakeUnionFind}$ in time $O(n)$ where $n = |S|$.
- For an element $u \in S$, the operation $\text{Find}(u)$ will return the name of the set containing $u$. Our goal will be to implement $\text{Find}(u)$ in $O(\log n)$ time. Some implementations that we discuss will in fact take only $O(1)$ time for this operation.
- For two sets $A$ and $B$, the operation $\text{Union}(A, B)$ will change the data structure by merging the sets $A$ and $B$ into a single set. Our goal will be to implement $\text{Union}$ in $O(\log n)$ time.

Let’s briefly discuss what we mean by the name of a set—for example, as returned by the $\text{Find}$ operation. There is a fair amount of flexibility in defining the names of the sets; they should simply be consistent in the sense that $\text{Find}(v)$ and $\text{Find}(w)$ should return the same name if $v$ and $w$ belong to the same set, and different names otherwise. In our implementations, we will name each set using one of the elements it contains.

### A Simple Data Structure for Union-Find

Maybe the simplest possible way to implement a Union-Find data structure is to maintain an array $\text{Component}$ that contains the name of the set currently containing each element. Let $S$ be a set, and assume it has $n$ elements denoted $\{1, \ldots, n\}$. We will set up an array $\text{Component}$ of size $n$, where $\text{Component}[s]$ is the name of the set containing $s$. To implement $\text{MakeUnionFind}(S)$, we set up the array and initialize it to $\text{Component}[s] = s$ for all $s \in S$. This implementation makes $\text{Find}(v)$ easy: it is a simple lookup and takes only $O(1)$ time. However, $\text{Union}(A, B)$ for two sets $A$ and $B$ can take as long as $O(n)$ time, as we have to update the values of $\text{Component}[s]$ for all elements in sets $A$ and $B$.

To improve this bound, we will do a few simple optimizations. First, it is useful to explicitly maintain the list of elements in each set, so we don’t have to look through the whole array to find the elements that need updating. Further,
we save some time by choosing the name for the union to be the name of one of the sets, say, set $A$: this way we only have to update the values $\text{Component}[s]$ for $s \in B$, but not for any $s \in A$. Of course, if set $B$ is large, this idea by itself doesn’t help very much. Thus we add one further optimization. When set $B$ is big, we may want to keep its name and change $\text{Component}[s]$ for all $s \in A$ instead. More generally, we can maintain an additional array $\text{size}$ of length $n$, where $\text{size}[A]$ is the size of set $A$, and when a $\text{Union}(A, B)$ operation is performed, we use the name of the larger set for the union. This way, fewer elements need to have their $\text{Component}$ values updated.

Even with these optimizations, the worst case for a $\text{Union}$ operation is still $O(n)$ time; this happens if we take the union of two large sets $A$ and $B$, each containing a constant fraction of all the elements. However, such bad cases for $\text{Union}$ cannot happen very often, as the resulting set $A \cup B$ is even bigger. How can we make this statement more precise? Instead of bounding the worst-case running time of a single $\text{Union}$ operation, we can bound the total (or average) running time of a sequence of $k$ $\text{Union}$ operations.

(4.23) Consider the array implementation of the Union-Find data structure for some set $S$ of size $n$, where unions keep the name of the larger set. The $\text{Find}$ operation takes $O(1)$ time, $\text{MakeUnionFind}(S)$ takes $O(n)$ time, and any sequence of $k$ $\text{Union}$ operations takes at most $O(k \log k)$ time.

Proof. The claims about the $\text{MakeUnionFind}$ and $\text{Find}$ operations are easy to verify. Now consider a sequence of $k$ $\text{Union}$ operations. The only part of a $\text{Union}$ operation that takes more than $O(1)$ time is updating the array $\text{Component}$. Instead of bounding the time spent on one $\text{Union}$ operation, we will bound the total time spent updating $\text{Component}[v]$ for an element $v$ throughout the sequence of $k$ operations.

Recall that we start the data structure from a state when all $n$ elements are in their own separate sets. A single $\text{Union}$ operation can consider at most two of these original one-element sets, so after any sequence of $k$ $\text{Union}$ operations, all but at most $2k$ elements of $S$ have been completely untouched. Now consider a particular element $v$. As $v$’s set is involved in a sequence of $\text{Union}$ operations, its size grows. It may be that in some of these $\text{Unions}$, the value of $\text{Component}[v]$ is updated, and in others it is not. But our convention is that the union uses the name of the larger set, so in every update to $\text{Component}[v]$ the size of the set containing $v$ at least doubles. The size of $v$’s set starts out at 1, and the maximum possible size it can reach is $2k$ (since we argued above that all but at most $2k$ elements are untouched by $\text{Union}$ operations). Thus $\text{Component}[v]$ gets updated at most $\log_2(2k)$ times throughout the process. Moreover, at most $2k$ elements are involved in any $\text{Union}$ operations at all, so
Chapter 4  Greedy Algorithms

we get a bound of $O(k \log k)$ for the time spent updating Component values in a sequence of $k$ Union operations. □

While this bound on the average running time for a sequence of $k$ operations is good enough in many applications, including implementing Kruskal’s Algorithm, we will try to do better and reduce the worst-case time required. We’ll do this at the expense of raising the time required for the Find operation to $O(\log n)$.

A Better Data Structure for Union-Find

The data structure for this alternate implementation uses pointers. Each node $v \in S$ will be contained in a record with an associated pointer to the name of the set that contains $v$. As before, we will use the elements of the set $S$ as possible set names, naming each set after one of its elements. For the MakeUnionFind($S$) operation, we initialize a record for each element $v \in S$ with a pointer that points to itself (or is defined as a null pointer), to indicate that $v$ is in its own set.

Consider a Union operation for two sets $A$ and $B$, and assume that the name we used for set $A$ is a node $v \in A$, while set $B$ is named after node $u \in B$. The idea is to have either $u$ or $v$ be the name of the combined set; assume we select $v$ as the name. To indicate that we took the union of the two sets, and that the name of the union set is $v$, we simply update $u$’s pointer to point to $v$. We do not update the pointers at the other nodes of set $B$.

As a result, for elements $w \in B$ other than $u$, the name of the set they belong to must be computed by following a sequence of pointers, first leading them to the “old name” $u$ and then via the pointer from $u$ to the “new name” $v$. See Figure 4.12 for what such a representation looks like. For example, the two sets in Figure 4.12 could be the outcome of the following sequence of Union operations: Union($w$, $u$), Union($s$, $u$), Union($t$, $v$), Union($z$, $v$), Union($i$, $x$), Union($y$, $j$), Union($x$, $j$), and Union($u$, $v$).

This pointer-based data structure implements Union in $O(1)$ time: all we have to do is to update one pointer. But a Find operation is no longer constant time, as we have to follow a sequence of pointers through a history of old names the set had, in order to get to the current name. How long can a Find($u$) operation take? The number of steps needed is exactly the number of times the set containing node $u$ had to change its name, that is, the number of times the Component[$u$] array position would have been updated in our previous array representation. This can be as large as $O(n)$ if we are not careful with choosing set names. To reduce the time required for a Find operation, we will use the same optimization we used before: keep the name of the larger set as the name of the union. The sequence of Unions that produced the data
The set \{s, u, w\} was merged into \{t, v, z\}.

Figure 4.12 A Union–Find data structure using pointers. The data structure has only two sets at the moment, named after nodes \(v\) and \(j\). The dashed arrow from \(u\) to \(v\) is the result of the last Union operation. To answer a Find query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query \(\text{Find}(i)\) would involve following the arrows \(i\) to \(x\), and then \(x\) to \(j\).

structure in Figure 4.12 followed this convention. To implement this choice efficiently, we will maintain an additional field with the nodes: the size of the corresponding set.

(4.24) Consider the above pointer-based implementation of the Union–Find data structure for some set \(S\) of size \(n\), where unions keep the name of the larger set. A Union operation takes \(O(1)\) time, MakeUnionFind\((S)\) takes \(O(n)\) time, and a Find operation takes \(O(\log n)\) time.

Proof. The statements about Union and MakeUnionFind are easy to verify. The time to evaluate Find\((v)\) for a node \(v\) is the number of times the set containing node \(v\) changes its name during the process. By the convention that the union keeps the name of the larger set, it follows that every time the name of the set containing node \(v\) changes, the size of this set at least doubles. Since the set containing \(v\) starts at size 1 and is never larger than \(n\), its size can double at most \(\log_2 n\) times, and so there can be at most \(\log_2 n\) name changes.

Further Improvements

Next we will briefly discuss a natural optimization in the pointer-based Union–Find data structure that has the effect of speeding up the Find operations. Strictly speaking, this improvement will not be necessary for our purposes in this book: for all the applications of Union–Find data structures that we consider, the \(O(\log n)\) time per operation is good enough in the sense that further improvement in the time for operations would not translate to improvements.
in the overall running time of the algorithms where we use them. (The Union-Find operations will not be the only computational bottleneck in the running time of these algorithms.)

To motivate the improved version of the data structure, let us first discuss a bad case for the running time of the pointer-based Union-Find data structure. First we build up a structure where one of the Find operations takes about \( \log n \) time. To do this, we can repeatedly take Unions of equal-sized sets. Assume \( v \) is a node for which the Find(\( v \)) operation takes about \( \log n \) time. Now we can issue Find(\( v \)) repeatedly, and it takes \( \log n \) for each such call. Having to follow the same sequence of \( \log n \) pointers every time for finding the name of the set containing \( v \) is quite redundant: after the first request for Find(\( v \)), we already “know” the name \( x \) of the set containing \( v \), and we also know that all other nodes that we touched during our path from \( v \) to the current name also are all contained in the set \( x \). So in the improved implementation, we will compress the path we followed after every Find operation by resetting all pointers along the path to point to the current name of the set. No information is lost by doing this, and it makes subsequent Find operations run more quickly. See Figure 4.13 for a Union-Find data structure and the result of Find(\( v \)) using path compression.

Now consider the running time of the operations in the resulting implementation. As before, a Union operation takes \( O(1) \) time and MakeUnion-Find(\( S \)) takes \( O(n) \) time to set up a data structure for a set of size \( n \). How did the time required for a Find(\( v \)) operation change? Some Find operations can still take up to \( \log n \) time; and for some Find operations we actually increase

![Figure 4.13](a) An instance of a Union-Find data structure; and (b) the result of the operation Find(\( v \)) on this structure, using path compression.

For any query, contact us at help@tutorialsduniya.com
the time, since after finding the name $x$ of the set containing $v$, we have to go back through the same path of pointers from $v$ to $x$, and reset each of these pointers to point to $x$ directly. But this additional work can at most double the time required, and so does not change the fact that a Find takes at most $O(\log n)$ time. The real gain from compression is in making subsequent calls to Find cheaper, and this can be made precise by the same type of argument we used in (4.23): bounding the total time for a sequence of $n$ Find operations, rather than the worst-case time for any one of them. Although we do not go into the details here, a sequence of $n$ Find operations employing compression requires an amount of time that is extremely close to linear in $n$; the actual upper bound is $O(n \alpha(n))$, where $\alpha(n)$ is an extremely slow-growing function of $n$ called the inverse Ackermann function. (In particular, $\alpha(n) \leq 4$ for any value of $n$ that could be encountered in practice.)

**Implementing Kruskal’s Algorithm**

Now we’ll use the Union–Find data structure to implement Kruskal’s Algorithm. First we need to sort the edges by cost. This takes time $O(m \log m)$. Since we have at most one edge between any pair of nodes, we have $m \leq n^2$ and hence this running time is also $O(m \log n)$.

After the sorting operation, we use the Union–Find data structure to maintain the connected components of $(V, T)$ as edges are added. As each edge $e = (v, w)$ is considered, we compute $\text{Find}(u)$ and $\text{Find}(v)$ and test if they are equal to see if $v$ and $w$ belong to different components. We use $\text{Union}$$\left(\text{Find}(u), \text{Find}(v)\right)$ to merge the two components, if the algorithm decides to include edge $e$ in the tree $T$.

We are doing a total of at most $2m$ Find and $n - 1$ Union operations over the course of Kruskal’s Algorithm. We can use either (4.23) for the array-based implementation of Union–Find, or (4.24) for the pointer-based implementation, to conclude that this is a total of $O(m \log n)$ time. (While more efficient implementations of the Union–Find data structure are possible, this would not help the running time of Kruskal’s Algorithm, which has an unavoidable $O(m \log n)$ term due to the initial sorting of the edges by cost.)

To sum up, we have

(4.25) **Kruskal’s Algorithm can be implemented on a graph with $n$ nodes and $m$ edges to run in $O(m \log n)$ time.**
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

<table>
<thead>
<tr>
<th>Subject</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>Computer System Architecture</td>
</tr>
<tr>
<td>Programming in Java</td>
<td>Discrete Structures</td>
</tr>
<tr>
<td>Data Structures</td>
<td>Operating Systems</td>
</tr>
<tr>
<td>Computer Networks</td>
<td>Algorithms</td>
</tr>
<tr>
<td>Android Programming</td>
<td>DataBase Management Systems</td>
</tr>
<tr>
<td>PHP Programming</td>
<td>Software Engineering</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Theory of Computation</td>
</tr>
<tr>
<td>Java Server Pages</td>
<td>Operational Research</td>
</tr>
<tr>
<td>Python</td>
<td>System Programming</td>
</tr>
<tr>
<td>Microprocessor</td>
<td>Data Mining</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
<td>Computer Graphics</td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Data Science</td>
</tr>
</tbody>
</table>

- Programs: [https://www.tutorialsduniya.com/programs](https://www.tutorialsduniya.com/programs)
- Question Papers: [https://www.tutorialsduniya.com/question-papers](https://www.tutorialsduniya.com/question-papers)
- Python Notes: [https://www.tutorialsduniya.com/python](https://www.tutorialsduniya.com/python)
- Java Notes: [https://www.tutorialsduniya.com/java](https://www.tutorialsduniya.com/java)
- JavaScript Notes: [https://www.tutorialsduniya.com/javascript](https://www.tutorialsduniya.com/javascript)
- JSP Notes: [https://www.tutorialsduniya.com/jsp](https://www.tutorialsduniya.com/jsp)
- Microprocessor Notes: [https://www.tutorialsduniya.com/microprocessor](https://www.tutorialsduniya.com/microprocessor)
- OR Notes: [https://www.tutorialsduniya.com/operational-research](https://www.tutorialsduniya.com/operational-research)
Suppose that three of your friends, inspired by repeated viewings of the horror-movie phenomenon *The Blair Witch Project*, have decided to hike the Appalachian Trail this summer. They want to hike as much as possible per day but, for obvious reasons, not after dark. On a map they’ve identified a large set of good *stopping points* for camping, and they’re considering the following system for deciding when to stop for the day. Each time they come to a potential stopping point, they determine whether they can make it to the next one before nightfall. If they can make it, then they keep hiking; otherwise, they stop.

Despite many significant drawbacks, they claim this system does have one good feature. “Given that we’re only hiking in the daylight,” they claim, “it minimizes the number of camping stops we have to make.”

Is this true? The proposed system is a greedy algorithm, and we wish to determine whether it minimizes the number of stops needed.

To make this question precise, let’s make the following set of simplifying assumptions. We’ll model the Appalachian Trail as a long line segment of length $L$, and assume that your friends can hike $d$ miles per day (independent of terrain, weather conditions, and so forth). We’ll assume that the potential stopping points are located at distances $x_1, x_2, \ldots, x_n$ from the start of the trail. We’ll also assume (very generously) that your friends are always correct when they estimate whether they can make it to the next stopping point before nightfall.

We’ll say that a set of stopping points is *valid* if the distance between each adjacent pair is at most $d$, the first is at distance at most $d$ from the start of the trail, and the last is at distance at most $d$ from the end of the trail. Thus a set of stopping points is valid if one could camp only at these places and
still make it across the whole trail. We’ll assume, naturally, that the full set of \( n \) stopping points is valid; otherwise, there would be no way to make it the whole way.

We can now state the question as follows. Is your friends’ greedy algorithm—hiking as long as possible each day—optimal, in the sense that it finds a valid set whose size is as small as possible?

**Solution** Often a greedy algorithm looks correct when you first encounter it, so before succumbing too deeply to its intuitive appeal, it’s useful to ask: why might it not work? What should we be worried about?

There’s a natural concern with this algorithm: Might it not help to stop early on some day, so as to get better synchronized with camping opportunities on future days? But if you think about it, you start to wonder whether this could really happen. Could there really be an alternate solution that intentionally lags behind the greedy solution, and then puts on a burst of speed and passes the greedy solution? How could it pass it, given that the greedy solution travels as far as possible each day?

This last consideration starts to look like the outline of an argument based on the “staying ahead” principle from Section 4.1. Perhaps we can show that as long as the greedy camping strategy is ahead on a given day, no other solution can catch up and overtake it the next day.

We now turn this into a proof showing the algorithm is indeed optimal, identifying a natural sense in which the stopping points it chooses “stay ahead” of any other legal set of stopping points. Although we are following the style of proof from Section 4.1, it’s worth noting an interesting contrast with the Interval Scheduling Problem: there we needed to prove that a greedy algorithm maximized a quantity of interest, whereas here we seek to minimize a certain quantity.

Let \( R = \{x_{p1}, \ldots, x_{pk}\} \) denote the set of stopping points chosen by the greedy algorithm, and suppose by way of contradiction that there is a smaller valid set of stopping points; let’s call this smaller set \( S = \{x_{q1}, \ldots, x_{qm}\} \), with \( m < k \).

To obtain a contradiction, we first show that the stopping point reached by the greedy algorithm on each day \( j \) is farther than the stopping point reached under the alternate solution. That is,

\[
(4.40) \quad \text{For each } j = 1, 2, \ldots, m, \text{ we have } x_{pj} \geq x_{qj}.
\]

**Proof.** We prove this by induction on \( j \). The case \( j = 1 \) follows directly from the definition of the greedy algorithm: your friends travel as long as possible...
on the first day before stopping. Now let \( j > 1 \) and assume that the claim is true for all \( i < j \). Then

\[
x_{q_j} - x_{q_{j-1}} \leq d,
\]
since \( S \) is a valid set of stopping points, and

\[
x_{q_j} - x_{p_{j-1}} \leq x_{q_j} - x_{q_{j-1}}
\]
since \( x_{p_{j-1}} \geq x_{q_{j-1}} \) by the induction hypothesis. Combining these two inequalities, we have

\[
x_{q_j} - x_{p_{j-1}} \leq d.
\]

This means that your friends have the option of hiking all the way from \( x_{p_{j-1}} \) to \( x_{q_j} \) in one day; and hence the location \( x_{p_j} \) at which they finally stop can only be farther along than \( x_{q_j} \). (Note the similarity with the corresponding proof for the Interval Scheduling Problem: here too the greedy algorithm is staying ahead because, at each step, the choice made by the alternate solution is one of its valid options.)

Statement (4.40) implies in particular that \( x_{q_m} \leq x_{p_m} \). Now, if \( m < k \), then we must have \( x_{p_m} < L - d \), for otherwise your friends would never have needed to stop at the location \( x_{p_{m+1}} \). Combining these two inequalities, we have concluded that \( x_{q_m} < L - d \); but this contradicts the assumption that \( S \) is a valid set of stopping points.

Consequently, we cannot have \( m < k \), and so we have proved that the greedy algorithm produces a valid set of stopping points of minimum possible size.

**Solved Exercise 2**

Your friends are starting a security company that needs to obtain licenses for \( n \) different pieces of cryptographic software. Due to regulations, they can only obtain these licenses at the rate of at most one per month.

Each license is currently selling for a price of $100. However, they are all becoming more expensive according to exponential growth curves: in particular, the cost of license \( j \) increases by a factor of \( r_j > 1 \) each month, where \( r_j \) is a given parameter. This means that if license \( j \) is purchased \( t \) months from now, it will cost \( 100 \cdot r_j^t \). We will assume that all the price growth rates are distinct; that is, \( r_i \neq r_j \) for licenses \( i \neq j \) (even though they start at the same price of $100).
Chapter 4  Greedy Algorithms

The question is: Given that the company can only buy at most one license a month, in which order should it buy the licenses so that the total amount of money it spends is as small as possible?

Give an algorithm that takes the \( n \) rates of price growth \( r_1, r_2, \ldots, r_n \), and computes an order in which to buy the licenses so that the total amount of money spent is minimized. The running time of your algorithm should be polynomial in \( n \).

**Solution**  Two natural guesses for a good sequence would be to sort the \( r_i \) in decreasing order, or to sort them in increasing order. Faced with alternatives like this, it’s perfectly reasonable to work out a small example and see if the example eliminates at least one of them. Here we could try \( r_1 = 2 \), \( r_2 = 3 \), and \( r_3 = 4 \). Buying the licenses in increasing order results in a total cost of

\[
100(2 + 3^2 + 4^3) = 7,500,
\]

while buying them in decreasing order results in a total cost of

\[
100(4 + 3^2 + 2^3) = 2,100.
\]

This tells us that increasing order is not the way to go. (On the other hand, it doesn’t tell us immediately that decreasing order is the right answer, but our goal was just to eliminate one of the two options.)

Let’s try proving that sorting the \( r_i \) in decreasing order in fact always gives the optimal solution. When a greedy algorithm works for problems like this, in which we put a set of things in an optimal order, we’ve seen in the text that it’s often effective to try proving correctness using an exchange argument.

To do this here, let’s suppose that there is an optimal solution \( O \) that differs from our solution \( S \). (In other words, \( S \) consists of the licenses sorted in decreasing order.) So this optimal solution \( O \) must contain an inversion—that is, there must exist two neighboring months \( t \) and \( t + 1 \) such that the price increase rate of the license bought in month \( t \) (let us denote it by \( r_t \)) is less than that bought in month \( t + 1 \) (similarly, we use \( r_{t+1} \) to denote this). That is, we have \( r_t < r_{t+1} \).

We claim that by exchanging these two purchases, we can strictly improve our optimal solution, which contradicts the assumption that \( O \) was optimal. Therefore if we succeed in showing this, we will successfully show that our algorithm is indeed the correct one.

Notice that if we swap these two purchases, the rest of the purchases are identically priced. In \( O \), the amount paid during the two months involved in the swap is \( 100(r_t^t + r_{t+1}^{t+1}) \). On the other hand, if we swapped these two purchases, we would pay \( 100(r_{t+1}^t + r_t^{t+1}) \). Since the constant 100 is common...
to both expressions, we want to show that the second term is less than the first one. So we want to show that

\[ r_{t+1}^t + r_{t+1}^{t+1} < r_t^t + r_{t+1}^{t+1} \]
\[ r_t^{t+1} - r_t^t < r_{t+1}^{t+1} - r_{t+1}^t \]
\[ r_t^t(r_t - 1) < r_{t+1}^t(r_t + 1 - 1). \]

But this last inequality is true simply because \( r_i > 1 \) for all \( i \) and since \( r_t < r_{t+1} \).

This concludes the proof of correctness. The running time of the algorithm is \( O(n \log n) \), since the sorting takes that much time and the rest (outputting) is linear. So the overall running time is \( O(n \log n) \).

Note: It’s interesting to note that things become much less straightforward if we vary this question even a little. Suppose that instead of buying licenses whose prices increase, you’re trying to sell off equipment whose cost is depreciating. Item \( i \) depreciates at a factor of \( r_i < 1 \) per month, starting from $100, so if you sell it \( t \) months from now you will receive \( 100 \cdot r_t^i \). (In other words, the exponential rates are now less than 1, instead of greater than 1.) If you can only sell one item per month, what is the optimal order in which to sell them? Here, it turns out that there are cases in which the optimal solution doesn’t put the rates in either increasing or decreasing order (as in the input \( \frac{3}{4}, \frac{1}{2}, \frac{1}{100} \)).

**Solved Exercise 3**

Suppose you are given a connected graph \( G \), with edge costs that you may assume are all distinct. \( G \) has \( n \) vertices and \( m \) edges. A particular edge \( e \) of \( G \) is specified. Give an algorithm with running time \( O(m + n) \) to decide whether \( e \) is contained in a minimum spanning tree of \( G \).

**Solution**  From the text, we know of two rules by which we can conclude whether an edge \( e \) belongs to a minimum spanning tree: the Cut Property (4.17) says that \( e \) is in every minimum spanning tree when it is the cheapest edge crossing from some set \( S \) to the complement \( V - S \); and the Cycle Property (4.20) says that \( e \) is in no minimum spanning tree if it is the most expensive edge on some cycle \( C \). Let’s see if we can make use of these two rules as part of an algorithm that solves this problem in linear time.

Both the Cut and Cycle Properties are essentially talking about how \( e \) relates to the set of edges that are *cheaper* than \( e \). The Cut Property can be viewed as asking: Is there some set \( S \subseteq V \) so that in order to get from \( S \) to \( V - S \) without using \( e \), we need to use an edge that is more expensive than \( e \)? And if we think about the cycle \( C \) in the statement of the Cycle Property, going the
Chapter 4  Greedy Algorithms

“long way” around C (avoiding e) can be viewed as an alternate route between the endpoints of e that only uses cheaper edges.

Putting these two observations together suggests that we should try proving the following statement.

\[(4.41)\] Edge \(e = (v, w)\) does not belong to a minimum spanning tree of \(G\) if and only if \(v\) and \(w\) can be joined by a path consisting entirely of edges that are cheaper than \(e\).

**Proof.** First suppose that \(P\) is a \(v\)-\(w\) path consisting entirely of edges cheaper than \(e\). If we add \(e\) to \(P\), we get a cycle on which \(e\) is the most expensive edge. Thus, by the Cycle Property, \(e\) does not belong to a minimum spanning tree of \(G\).

On the other hand, suppose that \(v\) and \(w\) cannot be joined by a path consisting entirely of edges cheaper than \(e\). We will now identify a set \(S\) for which \(e\) is the cheapest edge with one end in \(S\) and the other in \(V - S\); if we can do this, the Cut Property will imply that \(e\) belongs to every minimum spanning tree. Our set \(S\) will be the set of all nodes that are reachable from \(v\) using a path consisting only of edges that are cheaper than \(e\). By our assumption, we have \(w \in V - S\). Also, by the definition of \(S\), there cannot be an edge \(f = (x, y)\) that is cheaper than \(e\), and for which one end \(x\) lies in \(S\) and the other end \(y\) lies in \(V - S\). Indeed, if there were such an edge \(f\), then since the node \(x\) is reachable from \(v\) using only edges cheaper than \(e\), the node \(y\) would be reachable as well. Hence \(e\) is the cheapest edge with one end in \(S\) and the other in \(V - S\), as desired, and so we are done.

Given this fact, our algorithm is now simply the following. We form a graph \(G'\) by deleting from \(G\) all edges of weight greater than \(c_e\) (as well as deleting \(e\) itself). We then use one of the connectivity algorithms from Chapter 3 to determine whether there is a path from \(v\) to \(w\) in \(G'\). Statement (4.41) says that \(e\) belongs to a minimum spanning tree if and only if there is no such path.

The running time of this algorithm is \(O(m + n)\) to build \(G'\), and \(O(m + n)\) to test for a path from \(v\) to \(w\).

**Exercises**

1. Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

   Let \(G\) be an arbitrary connected, undirected graph with a distinct cost \(c(e)\) on every edge \(e\). Suppose \(e^*\) is the cheapest edge in \(G\); that is, \(c(e^*) < c(e)\) for every
edge $e \neq e^*$. Then there is a minimum spanning tree $T$ of $G$ that contains the edge $e^*$.

2. For each of the following two statements, decide whether it is true or false. If it is true, give a short explanation. If it is false, give a counterexample.

(a) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph $G$, with edge costs that are all positive and distinct. Let $T$ be a minimum spanning tree for this instance. Now suppose we replace each edge cost $c_e$ by its square, $c_e^2$, thereby creating a new instance of the problem with the same graph but different costs.

True or false? $T$ must still be a minimum spanning tree for this new instance.

(b) Suppose we are given an instance of the Shortest $s$-$t$ Path Problem on a directed graph $G$. We assume that all edge costs are positive and distinct. Let $P$ be a minimum-cost $s$-$t$ path for this instance. Now suppose we replace each edge cost $c_e$ by its square, $c_e^2$, thereby creating a new instance of the problem with the same graph but different costs.

True or false? $P$ must still be a minimum-cost $s$-$t$ path for this new instance.

3. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit $W$ on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package $i$ has a weight $w_i$. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.
Chapter 4 Greedy Algorithms

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it “stays ahead” of all other solutions.

4. Some of your friends have gotten into the burgeoning field of time-series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what’s being bought—are one source of data with a natural ordering in time. Given a long sequence $S$ of such events, your friends want an efficient way to detect certain “patterns” in them—for example, they may want to know if the four events

\[
\text{buy Yahoo, buy eBay, buy Yahoo, buy Oracle}
\]

occur in this sequence $S$, in order but not necessarily consecutively.

They begin with a collection of possible events (e.g., the possible transactions) and a sequence $S$ of $n$ of these events. A given event may occur multiple times in $S$ (e.g., Yahoo stock may be bought many times in a single sequence $S$). We will say that a sequence $S'$ is a subsequence of $S$ if there is a way to delete certain of the events from $S$ so that the remaining events, in order, are equal to the sequence $S'$. So, for example, the sequence of four events above is a subsequence of the sequence

\[
\text{buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle}
\]

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of $S$. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—$S'$ of length $m$ and $S$ of length $n$, each possibly containing an event more than once—and decides in time $O(m + n)$ whether $S'$ is a subsequence of $S$.

5. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.
6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected swimming time (the expected time it will take him or her to complete the 20 laps), a projected biking time (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected running time (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a schedule for the triathalon: an order in which to sequence the starts of the contestants. Let’s say that the completion time of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What’s the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

7. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They’ve broken the overall computation into \( n \) distinct jobs, labeled \( J_1, J_2, \ldots, J_n \), which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let’s say that job \( J_i \) needs \( p_i \) seconds of time on the supercomputer, followed by \( f_i \) seconds of time on a PC.

Since there are at least \( n \) PCs available on the premises, the finishing of the jobs can be performed fully in parallel—all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job
in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

8. Suppose you are given a connected graph $G$, with edge costs that are all distinct. Prove that $G$ has a unique minimum spanning tree.

9. One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible.

Specifically, let $G = (V, E)$ be a connected graph with $n$ vertices, $m$ edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of $G$; we define the bottleneck edge of $T$ to be the edge of $T$ with the greatest cost.

A spanning tree $T$ of $G$ is a minimum-bottleneck spanning tree if there is no spanning tree $T'$ of $G$ with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of $G$ a minimum spanning tree of $G$? Prove or give a counterexample.

(b) Is every minimum spanning tree of $G$ a minimum-bottleneck tree of $G$? Prove or give a counterexample.

10. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree $T$ in $G$. Now assume that a new edge is added to $G$, connecting two nodes $v, w \in V$ with cost $c$.

(a) Give an efficient algorithm to test if $T$ remains the minimum-cost spanning tree with the new edge added to $G$ (but not to the tree $T$). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree $T$ and the graph $G$.
(b) Suppose $T$ is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree $T$ to the new minimum-cost spanning tree.

11. Suppose you are given a connected graph $G = (V, E)$, with a cost $c_e$ on each edge $e$. In an earlier problem, we saw that when all edge costs are distinct, $G$ has a unique minimum spanning tree. However, $G$ may have many minimum spanning trees when the edge costs are not all distinct. Here we formulate the question: Can Kruskal’s Algorithm be made to find all the minimum spanning trees of $G$?

Recall that Kruskal’s Algorithm sorted the edges in order of increasing cost, then greedily processed edges one by one, adding an edge $e$ as long as it did not form a cycle. When some edges have the same cost, the phrase “in order of increasing cost” has to be specified a little more carefully: we'll say that an ordering of the edges is valid if the corresponding sequence of edge costs is nondecreasing. We'll say that a valid execution of Kruskal’s Algorithm is one that begins with a valid ordering of the edges of $G$.

For any graph $G$, and any minimum spanning tree $T$ of $G$, is there a valid execution of Kruskal’s Algorithm on $G$ that produces $T$ as output? Give a proof or a counterexample.

12. Suppose you have $n$ video streams that need to be sent, one after another, over a communication link. Stream $i$ consists of a total of $b_i$ bits that need to be sent, at a constant rate, over a period of $t_i$ seconds. You cannot send two streams at the same time, so you need to determine a schedule for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time $0$ (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values $b_i$ and $t_i$ are positive integers.

Now, because you’re just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter $r$:

\((*\) For each natural number $t > 0$, the total number of bits you send over the time interval from $0$ to $t$ cannot exceed $rt$.

Note that this constraint is only imposed for time intervals that start at $0$, not for time intervals that start at any other value.

We say that a schedule is valid if it satisfies the constraint $\ast$ imposed by the link.
Chapter 4 Greedy Algorithms

The Problem. Given a set of $n$ streams, each specified by its number of bits $b_i$ and its time duration $t_i$, as well as the link parameter $r$, determine whether there exists a valid schedule.

Example. Suppose we have $n = 3$ streams, with

\[
(b_1, t_1) = (2000, 1), \quad (b_2, t_2) = (6000, 2), \quad (b_3, t_3) = (2000, 1),
\]

and suppose the link’s parameter is $r = 5000$. Then the schedule that runs the streams in the order 1, 2, 3, is valid, since the constraint (*) is satisfied:

- $t = 1$: the whole first stream has been sent, and $2000 < 5000 \cdot 1$
- $t = 2$: half of the second stream has also been sent, and $2000 + 3000 < 5000 \cdot 2$

Similar calculations hold for $t = 3$ and $t = 4$.

(a) Consider the following claim:

Claim: There exists a valid schedule if and only if each stream $i$ satisfies $b_i \leq rt_i$.

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

(b) Give an algorithm that takes a set of $n$ streams, each specified by its number of bits $b_i$ and its time duration $t_i$, as well as the link parameter $r$, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in $n$.

13. A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer $i$’s job will take $t_i$ time to complete. Given a schedule (i.e., an ordering of the jobs), let $C_i$ denote the finishing time of job $i$. For example, if job $j$ is the first to be done, we would have $C_j = t_j$; and if job $j$ is done right after job $i$, we would have $C_j = C_i + t_j$. Each customer $i$ also has a given weight $w_i$ that represents his or her importance to the business. The happiness of customer $i$ is expected to be dependent on the finishing time of $i$’s job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of $n$ jobs with a processing time $t_i$ and a weight $w_i$ for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Example. Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 5$.
$w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

14. You're working with a group of security consultants who are helping to monitor a large computer system. There's particular interest in keeping track of processes that are labeled "sensitive." Each such process has a designated start time and finish time, and it runs continuously between these times; the consultants have a list of the planned start and finish times of all sensitive processes that will be run that day.

As a simple first step, they've written a program called status_check that, when invoked, runs for a few seconds and records various pieces of logging information about all the sensitive processes running on the system at that moment. (We'll model each invocation of status_check as lasting for only this single point in time.) What they'd like to do is to run status_check as few times as possible during the day, but enough that for each sensitive process $P$, status_check is invoked at least once during the execution of process $P$.

(a) Give an efficient algorithm that, given the start and finish times of all the sensitive processes, finds as small a set of times as possible at which to invoke status_check, subject to the requirement that status_check is invoked at least once during each sensitive process $P$.

(b) While you were designing your algorithm, the security consultants were engaging in a little back-of-the-envelope reasoning. "Suppose we can find a set of $k$ sensitive processes with the property that no two are ever running at the same time. Then clearly your algorithm will need to invoke status_check at least $k$ times: no one invocation of status_check can handle more than one of these processes."

This is true, of course, and after some further discussion, you all begin wondering whether something stronger is true as well, a kind of converse to the above argument. Suppose that $k^*$ is the largest value of $k$ such that one can find a set of $k$ sensitive processes with no two ever running at the same time. Is it the case that there must be a set of $k^*$ times at which you can run status_check so that some invocation occurs during the execution of each sensitive process? (In other words, the kind of argument in the previous paragraph is really the only thing forcing you to need a lot of invocations of status_check.) Decide whether you think this claim is true or false, and give a proof or a counterexample.
Chapter 4  Greedy Algorithms

15. The manager of a large student union on campus comes to you with the following problem. She's in charge of a group of $n$ students, each of whom is scheduled to work one shift during the week. There are different jobs associated with these shifts (tending the main desk, helping with package delivery, rebooting cranky information kiosks, etc.), but we can view each shift as a single contiguous interval of time. There can be multiple shifts going on at once.

She’s trying to choose a subset of these $n$ students to form a supervising committee that she can meet with once a week. She considers such a committee to be complete if, for every student not on the committee, that student's shift overlaps (at least partially) the shift of some student who is on the committee. In this way, each student’s performance can be observed by at least one person who’s serving on the committee.

Give an efficient algorithm that takes the schedule of $n$ shifts and produces a complete supervising committee containing as few students as possible.

Example. Suppose $n = 3$, and the shifts are

- Monday 4 p.m.–Monday 8 p.m.,
- Monday 6 p.m.–Monday 10 p.m.,
- Monday 9 p.m.–Monday 11 p.m.

Then the smallest complete supervising committee would consist of just the second student, since the second shift overlaps both the first and the third.

16. Some security consultants working in the financial domain are currently advising a client who is investigating a potential money-laundering scheme. The investigation thus far has indicated that $n$ suspicious transactions took place in recent days, each involving money transferred into a single account. Unfortunately, the sketchy nature of the evidence to date means that they don’t know the identity of the account, the amounts of the transactions, or the exact times at which the transactions took place. What they do have is an approximate time-stamp for each transaction; the evidence indicates that transaction $i$ took place at time $t_i \pm e_i$, for some “margin of error” $e_i$ (In other words, it took place sometime between $t_i - e_i$ and $t_i + e_i$.) Note that different transactions may have different margins of error.

In the last day or so, they've come across a bank account that (for other reasons we don’t need to go into here) they suspect might be the one involved in the crime. There are $n$ recent events involving the account, which took place at times $x_1, x_2, \ldots, x_n$. To see whether it’s plausible that this really is the account they’re looking for, they’re wondering
whether it’s possible to associate each of the account’s $n$ events with a distinct one of the $n$ suspicious transactions in such a way that, if the account event at time $x_i$ is associated with the suspicious transaction that occurred approximately at time $t_j$, then $|t_j - x_i| \leq e_j$. (In other words, they want to know if the activity on the account lines up with the suspicious transactions to within the margin of error; the tricky part here is that they don’t know which account event to associate with which suspicious transaction.)

Give an efficient algorithm that takes the given data and decides whether such an association exists. If possible, you should make the running time be at most $O(n^2)$.

17. Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of $n$ such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in $n$. You may assume for simplicity that no two jobs have the same start or end times.

**Example.** Consider the following four jobs, specified by (start-time, end-time) pairs.

$$(6 \text{ P.M.}, 6 \text{ A.M.}), (9 \text{ P.M.}, 4 \text{ A.M.}), (3 \text{ A.M.}, 2 \text{ P.M.}), (1 \text{ P.M.}, 7 \text{ P.M.}).$$

The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

18. Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They’ve researched all the travel options and have drawn up a directed graph whose nodes represent intermediate destinations and edges represent the roads between them.

In the course of this, they’ve also learned that extreme weather causes roads in this part of the world to become quite slow in the winter and may cause large travel delays. They’ve found an excellent travel Web site that can accurately predict how fast they’ll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an
edge $e = (v, w)$ connecting two sites $v$ and $w$, and given a proposed starting time $t$ from location $v$, the site will return a value $f_e(t)$, the predicted arrival time at $w$. The Web site guarantees that $f_e(t) \geq t$ for all edges $e$ and all times $t$ (you can’t travel backward in time), and that $f_e(t)$ is a monotone increasing function of $t$ (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + \ell_e$, where $\ell_e$ is the time needed to travel from the beginning to the end of edge $e$.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge $e$ and a time $t$) as taking a single computational step.

19. A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph $G = (V, E)$, in which the nodes represent sites that want to communicate. Each edge $e$ is a communication link, with a given available bandwidth $b_e$.

For each pair of nodes $u, v \in V$, they want to select a single $u$-$v$ path $P$ on which this pair will communicate. The bottleneck rate $b(P)$ of this path $P$ is the minimum bandwidth of any edge it contains; that is, $b(P) = \min_{e \in P} b_e$. The best achievable bottleneck rate for the pair $u, v$ in $G$ is simply the maximum, over all $u$-$v$ paths $P$ in $G$, of the value $b(P)$.

It’s getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can find a spanning tree $T$ of $G$ so that for every pair of nodes $u, v$, the unique $u$-$v$ path in the tree actually attains the best achievable bottleneck rate for $u, v$ in $G$. (In other words, even if you could choose any $u$-$v$ path in the whole graph, you couldn’t do better than the $u$-$v$ path in $T$.)

This idea is roundly heckled in the offices of CluNet for a few days, and there’s a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.
Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree \( T \) in which, for each \( u, v \in V \), the bottleneck rate of the \( u-v \) path in \( T \) is equal to the best achievable bottleneck rate for the pair \( u, v \) in \( G \).

20. Every September, somewhere in a far-away mountainous part of the world, the county highway crews get together and decide which roads to keep clear through the coming winter. There are \( n \) towns in this county, and the road system can be viewed as a (connected) graph \( G = (V, E) \) on this set of towns, each edge representing a road joining two of them. In the winter, people are high enough up in the mountains that they stop worrying about the length of roads and start worrying about their altitude — this is really what determines how difficult the trip will be.

So each road—each edge \( e \) in the graph—is annotated with a number \( a_e \) that gives the altitude of the highest point on the road. We’ll assume that no two edges have exactly the same altitude value \( a_e \). The height of a path \( P \) in the graph is then the maximum of \( a_e \) over all edges \( e \) on \( P \). Finally, a path between towns \( i \) and \( j \) is declared to be winter-optimal if it achieves the minimum possible height over all paths from \( i \) to \( j \).

The highway crews are going to select a set \( E' \subseteq E \) of the roads to keep clear through the winter; the rest will be left unmaintained and kept off limits to travelers. They all agree that whichever subset of roads \( E' \) they decide to keep clear, it should have the property that \( (V, E') \) is a connected subgraph; and more strongly, for every pair of towns \( i \) and \( j \), the height of the winter-optimal path in \( (V, E') \) should be no greater than it is in the full graph \( G = (V, E) \). We’ll say that \( (V, E') \) is a minimum-altitude connected subgraph if it has this property.

Given that they’re going to maintain this key property, however, they otherwise want to keep as few roads clear as possible. One year, they hit upon the following conjecture:

*The minimum spanning tree of \( G \), with respect to the edge weights \( a_e \), is a minimum-altitude connected subgraph.*

(In an earlier problem, we claimed that there is a unique minimum spanning tree when the edge weights are distinct. Thus, thanks to the assumption that all \( a_e \) are distinct, it is okay for us to speak of *the* minimum spanning tree.)

Initially, this conjecture is somewhat counterintuitive, since the minimum spanning tree is trying to minimize the sum of the values \( a_e \), while the goal of minimizing altitude seems to be asking for a fairly different thing. But lacking an argument to the contrary, they begin considering an even bolder second conjecture:
A subgraph \((V, E')\) is a minimum-altitude connected subgraph if and only if it contains the edges of the minimum spanning tree.

Note that this second conjecture would immediately imply the first one, since a minimum spanning tree contains its own edges.

So here’s the question.

(a) Is the first conjecture true, for all choices of \(G\) and distinct altitudes \(a_e\)? Give a proof or a counterexample with explanation.

(b) Is the second conjecture true, for all choices of \(G\) and distinct altitudes \(a_e\)? Give a proof or a counterexample with explanation.

21. Let us say that a graph \(G = (V, E)\) is a near-tree if it is connected and has at most \(n + 8\) edges, where \(n = |V|\). Give an algorithm with running time \(O(n)\) that takes a near-tree \(G\) with costs on its edges, and returns a minimum spanning tree of \(G\). You may assume that all the edge costs are distinct.

22. Consider the Minimum Spanning Tree Problem on an undirected graph \(G = (V, E)\), with a cost \(c_e \geq 0\) on each edge, where the costs may not all be different. If the costs are not all distinct, there can in general be many distinct minimum-cost solutions. Suppose we are given a spanning tree \(T \subseteq E\) with the guarantee that for every \(e \in T\), \(e\) belongs to some minimum-cost spanning tree in \(G\). Can we conclude that \(T\) itself must be a minimum-cost spanning tree in \(G\)? Give a proof or a counterexample with explanation.

23. Recall the problem of computing a minimum-cost arborescence in a directed graph \(G = (V, E)\), with a cost \(c_e \geq 0\) on each edge. Here we will consider the case in which \(G\) is a directed acyclic graph—that is, it contains no directed cycles.

As in general directed graphs, there can be many distinct minimum-cost solutions. Suppose we are given a directed acyclic graph \(G = (V, E)\), and an arborescence \(A \subseteq E\) with the guarantee that for every \(e \in A\), \(e\) belongs to some minimum-cost arborescence in \(G\). Can we conclude that \(A\) itself must be a minimum-cost arborescence in \(G\)? Give a proof or a counterexample with explanation.

24. Timing circuits are a crucial component of VLSI chips. Here’s a simple model of such a timing circuit. Consider a complete balanced binary tree with \(n\) leaves, where \(n\) is a power of two. Each edge \(e\) of the tree has an associated length \(\ell_e\), which is a positive number. The distance from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.
Figure 4.20 An instance of the zero-skew problem, described in Exercise 23.

The root generates a clock signal which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to increase the lengths of certain edges, so that all root-to-leaf paths have the same length (we’re not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have zero skew. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

Example. Consider the tree in Figure 4.20, in which letters name the nodes and numbers indicate the edge lengths.

The unique optimal solution for this instance would be to take the three length-1 edges and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

25. Suppose we are given a set of points \( P = \{p_1, p_2, \ldots, p_n\} \), together with a distance function \( d \) on the set \( P \); \( d \) is simply a function on pairs of points in \( P \) with the properties that \( d(p_i, p_j) = d(p_j, p_i) > 0 \) if \( i \neq j \), and that \( d(p_i, p_i) = 0 \) for each \( i \).

We define a hierarchical metric on \( P \) to be any distance function \( \tau \) that can be constructed as follows. We build a rooted tree \( T \) with \( n \) leaves, and we associate with each node \( v \) of \( T \) (both leaves and internal nodes) a height \( h_v \). These heights must satisfy the properties that \( h(v) = 0 \) for each
Chapter 4  Greedy Algorithms

leaf $v$, and if $u$ is the parent of $v$ in $T$, then $h(u) \geq h(v)$. We place each point in $P$ at a distinct leaf in $T$. Now, for any pair of points $p_i$ and $p_j$, their distance $\tau(p_i, p_j)$ is defined as follows. We determine the least common ancestor $v$ in $T$ of the leaves containing $p_i$ and $p_j$, and define $\tau(p_i, p_j) = h_v$.

We say that a hierarchical metric $\tau$ is consistent with our distance function $d$ if, for all pairs $i, j$, we have $\tau(p_i, p_j) \leq d(p_i, p_j)$.

Give a polynomial-time algorithm that takes the distance function $d$ and produces a hierarchical metric $\tau$ with the following properties.

(i) $\tau$ is consistent with $d$, and

(ii) if $\tau'$ is any other hierarchical metric consistent with $d$, then $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ for each pair of points $p_i$ and $p_j$.

26. One of the first things you learn in calculus is how to minimize a differentiable function such as $y = ax^2 + bx + c$, where $a > 0$. The Minimum Spanning Tree Problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved—rather than a continuum of possibilities—and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge $e$ now has a time-varying edge cost given by a function $f_e : \mathbb{R} \to \mathbb{R}$. Thus, at time $t$, it has cost $f_e(t)$. We’ll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of $G$ may change over time. Also, of course, the cost of the minimum spanning tree of $G$ becomes a function of the time $t$; we’ll denote this function $c_G(t)$. A natural problem then becomes: find a value of $t$ at which $c_G(t)$ is minimized.

Suppose each function $f_e$ is a polynomial of degree 2: $f_e(t) = a_et^2 + b_et + c_e$, where $a_e > 0$. Give an algorithm that takes the graph $G$ and the values $\{(a_e, b_e, c_e) : e \in E\}$ and returns a value of the time $t$ at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph $G$. You may assume that arithmetic operations on the numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

27. In trying to understand the combinatorial structure of spanning trees, we can consider the space of all possible spanning trees of a given graph and study the properties of this space. This is a strategy that has been applied to many similar problems as well.
Exercises

Here is one way to do this. Let $G$ be a connected graph, and $T$ and $T'$ two different spanning trees of $G$. We say that $T$ and $T'$ are neighbors if $T$ contains exactly one edge that is not in $T'$, and $T'$ contains exactly one edge that is not in $T$.

Now, from any graph $G$, we can build a (large) graph $H$ as follows. The nodes of $H$ are the spanning trees of $G$, and there is an edge between two nodes of $H$ if the corresponding spanning trees are neighbors.

Is it true that, for any connected graph $G$, the resulting graph $H$ is connected? Give a proof that $H$ is always connected, or provide an example (with explanation) of a connected graph $G$ for which $H$ is not connected.

28. Suppose you're a consultant for the networking company CluNet, and they have the following problem. The network that they're currently working on is modeled by a connected graph $G = (V, E)$ with $n$ nodes. Each edge $e$ is a fiber-optic cable that is owned by one of two companies—creatively named $X$ and $Y$—and leased to CluNet.

Their plan is to choose a spanning tree $T$ of $G$ and upgrade the links corresponding to the edges of $T$. Their business relations people have already concluded an agreement with companies $X$ and $Y$ stipulating a number $k$ so that in the tree $T$ that is chosen, $k$ of the edges will be owned by $X$ and $n - k - 1$ of the edges will be owned by $Y$.

CluNet management now faces the following problem. It is not at all clear to them whether there even exists a spanning tree $T$ meeting these conditions, or how to find one if it exists. So this is the problem they put to you: Give a polynomial-time algorithm that takes $G$, with each edge labeled $X$ or $Y$, and either (i) returns a spanning tree with exactly $k$ edges labeled $X$, or (ii) reports correctly that no such tree exists.

29. Given a list of $n$ natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (That is, if $V = \{v_1, v_2, \ldots, v_n\}$, then the degree of $v_i$ should be exactly $d_i$.) $G$ should not contain multiple edges between the same pair of nodes, or “loop” edges with both endpoints equal to the same node.

30. Let $G = (V, E)$ be a graph with $n$ nodes in which each pair of nodes is joined by an edge. There is a positive weight $w_{ij}$ on each edge $(i, j)$; and we will assume these weights satisfy the triangle inequality $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in $V'$. 

For any query, contact us at help@tutorialsduniya.com
Chapter 4  Greedy Algorithms

We are given a set \( X \subseteq V \) of \( k \) terminals that must be connected by edges. We say that a Steiner tree on \( X \) is a set \( Z \) so that \( X \subseteq Z \subseteq V \), together with a spanning subtree \( T \) of \( G[Z] \). The weight of the Steiner tree is the weight of the tree \( T \).

Show that the problem of finding a minimum-weight Steiner tree on \( X \) can be solved in time \( O(n^{O(k)}) \).

31. Let’s go back to the original motivation for the Minimum Spanning Tree Problem. We are given a connected, undirected graph \( G = (V, E) \) with positive edge lengths \( \{\ell_e\} \), and we want to find a spanning subgraph of it. Now suppose we are willing to settle for a subgraph \( H = (V, F) \) that is “denser” than a tree, and we are interested in guaranteeing that, for each pair of vertices \( u, v \in V \), the length of the shortest \( u\cdot v \) path in \( H \) is not much longer than the length of the shortest \( u\cdot v \) path in \( G \). By the length of a path \( P \) here, we mean the sum of \( \ell_e \) over all edges \( e \) in \( P \).

Here’s a variant of Kruskal’s Algorithm designed to produce such a subgraph.

- First we sort all the edges in order of increasing length. (You may assume all edge lengths are distinct.)
- We then construct a subgraph \( H = (V, F) \) by considering each edge in order.
- When we come to edge \( e = (u, v) \), we add \( e \) to the subgraph \( H \) if there is currently no \( u\cdot v \) path in \( H \). (This is what Kruskal’s Algorithm would do as well.) On the other hand, if there is a \( u\cdot v \) path in \( H \), we let \( d_{uv} \) denote the length of the shortest such path; again, length is with respect to the values \( \{\ell_e\} \). We add \( e \) to \( H \) if \( 3\ell_e < d_{uv} \).

In other words, we add an edge even when \( u \) and \( v \) are already in the same connected component, provided that the addition of the edge reduces their shortest-path distance by a sufficient amount.

Let \( H = (V, F) \) be the subgraph of \( G \) returned by the algorithm.

(a) Prove that for every pair of nodes \( u, v \in V \), the length of the shortest \( u\cdot v \) path in \( H \) is at most three times the length of the shortest \( u\cdot v \) path in \( G \).

(b) Despite its ability to approximately preserve shortest-path distances, the subgraph \( H \) produced by the algorithm cannot be too dense. Let \( f(n) \) denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all \( n \)-node input graphs with edge lengths. Prove that

\[
\lim_{n \to \infty} \frac{f(n)}{n^2} = 0.
\]
32. Consider a directed graph $G = (V, E)$ with a root $r \in V$ and nonnegative costs on the edges. In this problem we consider variants of the minimum-cost arborescence algorithm.

(a) The algorithm discussed in Section 4.9 works as follows. We modify the costs, consider the subgraph of zero-cost edges, look for a directed cycle in this subgraph, and contract it (if one exists). Argue briefly that instead of looking for cycles, we can instead identify and contract strong components of this subgraph.

(b) In the course of the algorithm, we defined $y_v$ to be the minimum cost of an edge entering $v$, and we modified the costs of all edges $e$ entering node $v$ to be $c'_e = c_e - y_v$. Suppose we instead use the following modified cost: $c''_e = \max(0, c_e - 2y_v)$. This new change is likely to turn more edges to 0 cost. Suppose now we find an arborescence $T$ of 0 cost. Prove that this $T$ has cost at most twice the cost of the minimum-cost arborescence in the original graph.

(c) Assume you do not find an arborescence of 0 cost. Contract all 0-cost strong components and recursively apply the same procedure on the resulting graph until an arborescence is found. Prove that this $T$ has cost at most twice the cost of the minimum-cost arborescence in the original graph.

33. Suppose you are given a directed graph $G = (V, E)$ in which each edge has a cost of either 0 or 1. Also suppose that $G$ has a node $r$ such that there is a path from $r$ to every other node in $G$. You are also given an integer $k$. Give a polynomial-time algorithm that either constructs an arborescence rooted at $r$ of cost exactly $k$, or reports (correctly) that no such arborescence exists.
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
</table>

TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
Chapter 6

Dynamic Programming

We began our study of algorithmic techniques with greedy algorithms, which in some sense form the most natural approach to algorithm design. Faced with a new computational problem, we’ve seen that it’s not hard to propose multiple possible greedy algorithms; the challenge is then to determine whether any of these algorithms provides a correct solution to the problem in all cases.

The problems we saw in Chapter 4 were all unified by the fact that, in the end, there really was a greedy algorithm that worked. Unfortunately, this is far from being true in general; for most of the problems that one encounters, the real difficulty is not in determining which of several greedy strategies is the right one, but in the fact that there is no natural greedy algorithm that works. For such problems, it is important to have other approaches at hand. Divide and conquer can sometimes serve as an alternative approach, but the versions of divide and conquer that we saw in the previous chapter are often not strong enough to reduce exponential brute-force search down to polynomial time. Rather, as we noted in Chapter 5, the applications there tended to reduce a running time that was unnecessarily large, but already polynomial, down to a faster running time.

We now turn to a more powerful and subtle design technique, dynamic programming. It will be easier to say exactly what characterizes dynamic programming after we’ve seen it in action, but the basic idea is drawn from the intuition behind divide and conquer and is essentially the opposite of the greedy strategy: one implicitly explores the space of all possible solutions, by carefully decomposing things into a series of subproblems, and then building up correct solutions to larger and larger subproblems. In a way, we can thus view dynamic programming as operating dangerously close to the edge of
brute-force search: although it’s systematically working through the exponentially large set of possible solutions to the problem, it does this without ever examining them all explicitly. It is because of this careful balancing act that dynamic programming can be a tricky technique to get used to; it typically takes a reasonable amount of practice before one is fully comfortable with it.

With this in mind, we now turn to a first example of dynamic programming: the Weighted Interval Scheduling Problem that we defined back in Section 1.2. We are going to develop a dynamic programming algorithm for this problem in two stages: first as a recursive procedure that closely resembles brute-force search; and then, by reinterpreting this procedure, as an iterative algorithm that works by building up solutions to larger and larger subproblems.

6.1 Weighted Interval Scheduling: A Recursive Procedure

We have seen that a particular greedy algorithm produces an optimal solution to the Interval Scheduling Problem, where the goal is to accept as large a set of nonoverlapping intervals as possible. The Weighted Interval Scheduling Problem is a strictly more general version, in which each interval has a certain value (or weight), and we want to accept a set of maximum value.

Designing a Recursive Algorithm

Since the original Interval Scheduling Problem is simply the special case in which all values are equal to 1, we know already that most greedy algorithms will not solve this problem optimally. But even the algorithm that worked before (repeatedly choosing the interval that ends earliest) is no longer optimal in this more general setting, as the simple example in Figure 6.1 shows.

Indeed, no natural greedy algorithm is known for this problem, which is what motivates our switch to dynamic programming. As discussed above, we will begin our introduction to dynamic programming with a recursive type of algorithm for this problem, and then in the next section we’ll move to a more iterative method that is closer to the style we use in the rest of this chapter.

Figure 6.1 A simple instance of weighted interval scheduling.

**Figure 6.1** A simple instance of weighted interval scheduling.
6.1 Weighted Interval Scheduling: A Recursive Procedure

We use the notation from our discussion of Interval Scheduling in Section 1.2. We have \( n \) requests labeled 1, \ldots, \( n \), with each request \( i \) specifying a start time \( s_i \) and a finish time \( f_i \). Each interval \( i \) now also has a value, or weight \( v_i \). Two intervals are compatible if they do not overlap. The goal of our current problem is to select a subset \( S \subseteq \{1, \ldots, n\} \) of mutually compatible intervals, so as to maximize the sum of the values of the selected intervals, \( \sum_{i \in S} v_i \).

Let’s suppose that the requests are sorted in order of nondecreasing finish time: \( f_1 \leq f_2 \leq \cdots \leq f_n \). We’ll say a request \( i \) comes before a request \( j \) if \( i < j \). This will be the natural left-to-right order in which we’ll consider intervals. To help in talking about this order, we define \( p(j) \), for an interval \( j \), to be the largest index \( i < j \) such that intervals \( i \) and \( j \) are disjoint. In other words, \( i \) is the leftmost interval that ends before \( j \) begins. We define \( p(j) = 0 \) if no request \( i < j \) is disjoint from \( j \). An example of the definition of \( p(j) \) is shown in Figure 6.2.

Now, given an instance of the Weighted Interval Scheduling Problem, let’s consider an optimal solution \( \mathcal{O} \), ignoring for now that we have no idea what it is. Here’s something completely obvious that we can say about \( \mathcal{O} \): either interval \( n \) (the last one) belongs to \( \mathcal{O} \), or it doesn’t. Suppose we explore both sides of this dichotomy a little further. If \( n \in \mathcal{O} \), then clearly no interval indexed strictly between \( p(n) \) and \( n \) can belong to \( \mathcal{O} \), because by the definition of \( p(n) \), we know that intervals \( p(n) + 1, p(n) + 2, \ldots, n - 1 \) all overlap interval \( n \). Moreover, if \( n \in \mathcal{O} \), then \( \mathcal{O} \) must include an optimal solution to the problem consisting of requests \( \{1, \ldots, p(n)\} \)—for if it didn’t, we could replace \( \mathcal{O} \)’s choice of requests from \( \{1, \ldots, p(n)\} \) with a better one, with no danger of overlapping request \( n \).

![Figure 6.2](https://www.tutorialsduniya.com) An instance of weighted interval scheduling with the functions \( p(j) \) defined for each interval \( j \).
On the other hand, if \( n \not\in \mathcal{O} \), then \( \mathcal{O} \) is simply equal to the optimal solution to the problem consisting of requests \( \{1, \ldots, n-1\} \). This is by completely analogous reasoning: we’re assuming that \( \mathcal{O} \) does not include request \( n \); so if it does not choose the optimal set of requests from \( \{1, \ldots, n-1\} \), we could replace it with a better one.

All this suggests that finding the optimal solution on intervals \( \{1, 2, \ldots, n\} \) involves looking at the optimal solutions of smaller problems of the form \( \{1, 2, \ldots, j\} \). Thus, for any value of \( j \) between 1 and \( n \), let \( \mathcal{O}_j \) denote the optimal solution to the problem consisting of requests \( \{1, \ldots, j\} \), and let \( \text{OPT}(j) \) denote the value of this solution. (We define \( \text{OPT}(0) = 0 \), based on the convention that this is the optimum over an empty set of intervals.) The optimal solution we’re seeking is precisely \( \mathcal{O}_n \), with value \( \text{OPT}(n) \). For the optimal solution \( \mathcal{O}_j \) on \( \{1, 2, \ldots, j\} \), our reasoning above (generalizing from the case in which \( j = n \)) says that either \( j \in \mathcal{O}_j \), in which case \( \text{OPT}(j) = v_j + \text{OPT}(p(j)) \), or \( j \not\in \mathcal{O}_j \), in which case \( \text{OPT}(j) = \text{OPT}(j-1) \). Since these are precisely the two possible choices \( (j \in \mathcal{O}_j \text{ or } j \not\in \mathcal{O}_j) \), we can further say that

\[
(6.1) \quad \text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)).
\]

And how do we decide whether \( n \) belongs to the optimal solution \( \mathcal{O}_j \)? This too is easy: it belongs to the optimal solution if and only if the first of the options above is at least as good as the second; in other words,

\[
(6.2) \quad \text{Request } j \text{ belongs to an optimal solution on the set } \{1, 2, \ldots, j\} \text{ if and only if } v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1).
\]

These facts form the first crucial component on which a dynamic programming solution is based: a recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller subproblems.

Despite the simple reasoning that led to this point, (6.1) is already a significant development. It directly gives us a recursive algorithm to compute \( \text{OPT}(n) \), assuming that we have already sorted the requests by finishing time and computed the values of \( p(j) \) for each \( j \).

\[
\text{Compute-Opt}(j)
\]
\[
\text{If } j = 0 \text{ then}
\]
\[
\text{Return } 0
\]
\[
\text{Else}
\]
\[
\text{Return } \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))
\]
\[
\text{Endif}
\]
The correctness of the algorithm follows directly by induction on $j$:

\[(6.3) \quad \text{Compute-Opt}(j) \text{ correctly computes } \text{OPT}(j) \text{ for each } j = 1, 2, \ldots, n.\]

**Proof.** By definition $\text{OPT}(0) = 0$. Now, take some $j > 0$, and suppose by way of induction that $\text{Compute-Opt}(i)$ correctly computes $\text{OPT}(i)$ for all $i < j$. By the induction hypothesis, we know that $\text{Compute-Opt}(p(j)) = \text{OPT}(p(j))$ and $\text{Compute-Opt}(j - 1) = \text{OPT}(j - 1)$; and hence from (6.1) it follows that

\[
\text{OPT}(j) = \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)) = \text{Compute-Opt}(j). \quad \blacksquare
\]

Unfortunately, if we really implemented the algorithm $\text{Compute-Opt}$ as just written, it would take exponential time to run in the worst case. For example, see Figure 6.3 for the tree of calls issued for the instance of Figure 6.2: the tree widens very quickly due to the recursive branching. To take a more extreme example, on a nicely layered instance like the one in Figure 6.4, where $p(j) = j - 2$ for each $j = 2, 3, 4, \ldots, n$, we see that $\text{Compute-Opt}(j)$ generates separate recursive calls on problems of sizes $j - 1$ and $j - 2$. In other words, the total number of calls made to $\text{Compute-Opt}$ on this instance will grow

![Figure 6.3](image-url)
like the Fibonacci numbers, which increase exponentially. Thus we have not achieved a polynomial-time solution.

### Memoizing the Recursion

In fact, though, we’re not so far from having a polynomial-time algorithm. A fundamental observation, which forms the second crucial component of a dynamic programming solution, is that our recursive algorithm Compute-Opt is really only solving $n + 1$ different subproblems: Compute-Opt(0), Compute-Opt(1), ..., Compute-Opt(n). The fact that it runs in exponential time as written is simply due to the spectacular redundancy in the number of times it issues each of these calls.

How could we eliminate all this redundancy? We could store the value of Compute-Opt in a globally accessible place the first time we compute it and then simply use this precomputed value in place of all future recursive calls. This technique of saving values that have already been computed is referred to as memoization.

We implement the above strategy in the more “intelligent” procedure M-Compute-Opt. This procedure will make use of an array $M[0 \ldots n]; M[j]$ will start with the value “empty,” but will hold the value of Compute-Opt(j) as soon as it is first determined. To determine OPT(n), we invoke M-Compute-Opt(n).

---

M-Compute-Opt(j)
If $j = 0$ then
Return 0
Else if $M[j]$ is not empty then
Return $M[j]$
Else
6.1 Weighted Interval Scheduling: A Recursive Procedure

Define $M[j] = \max(v_j + M\text{-Compute-Opt}(p(j)), M\text{-Compute-Opt}(j - 1))$

Return $M[j]$

Endif

Analyzing the Memoized Version

Clearly, this looks very similar to our previous implementation of the algorithm; however, memoization has brought the running time way down.

(6.4) The running time of $M\text{-Compute-Opt}(n)$ is $O(n)$ (assuming the input intervals are sorted by their finish times).

Proof. The time spent in a single call to $M\text{-Compute-Opt}$ is $O(1)$, excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls ever issued to $M\text{-Compute-Opt}$. Since the implementation itself gives no explicit upper bound on this number of calls, we try to find a bound by looking for a good measure of “progress.”

The most useful progress measure here is the number of entries in $M$ that are not “empty.” Initially this number is 0; but each time the procedure invokes the recurrence, issuing two recursive calls to $M\text{-Compute-Opt}$, it fills in a new entry, and hence increases the number of filled-in entries by 1. Since $M$ has only $n + 1$ entries, it follows that there can be at most $O(n)$ calls to $M\text{-Compute-Opt}$, and hence the running time of $M\text{-Compute-Opt}(n)$ is $O(n)$, as desired.

Computing a Solution in Addition to Its Value

So far we have simply computed the value of an optimal solution; presumably we want a full optimal set of intervals as well. It would be easy to extend $M\text{-Compute-Opt}$ so as to keep track of an optimal solution in addition to its value: we could maintain an additional array $S$ so that $S[i]$ contains an optimal set of intervals among $\{1, 2, \ldots, i\}$. Naively enhancing the code to maintain the solutions in the array $S$, however, would blow up the running time by an additional factor of $O(n)$: while a position in the $M$ array can be updated in $O(1)$ time, writing down a set in the $S$ array takes $O(n)$ time. We can avoid this $O(n)$ blow-up by not explicitly maintaining $S$, but rather by recovering the optimal solution from values saved in the array $M$ after the optimum value has been computed.

We know from (6.2) that $j$ belongs to an optimal solution for the set of intervals $\{1, \ldots, j\}$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$. Using this observation, we get the following simple procedure, which “traces back” through the array $M$ to find the set of intervals in an optimal solution.
Chapter 6 Dynamic Programming

Find-Solution\(j\)

If \(j = 0\) then

Output nothing

Else

If \(v_j + M[p(j)] \geq M[j - 1]\) then

Output \(j\) together with the result of Find-Solution\(p(j)\)

Else

Output the result of Find-Solution\(j - 1\)

Endif

Endif

Since Find-Solution calls itself recursively only on strictly smaller values, it makes a total of \(O(n)\) recursive calls; and since it spends constant time per call, we have

\(6.5\) Given the array \(M\) of the optimal values of the sub-problems, Find-Solution returns an optimal solution in \(O(n)\) time.

6.2 Principles of Dynamic Programming: Memoization or Iteration over Subproblems

We now use the algorithm for the Weighted Interval Scheduling Problem developed in the previous section to summarize the basic principles of dynamic programming, and also to offer a different perspective that will be fundamental to the rest of the chapter: iterating over subproblems, rather than computing solutions recursively.

In the previous section, we developed a polynomial-time solution to the Weighted Interval Scheduling Problem by first designing an exponential-time recursive algorithm and then converting it (by memoization) to an efficient recursive algorithm that consulted a global array \(M\) of optimal solutions to subproblems. To really understand what is going on here, however, it helps to formulate an essentially equivalent version of the algorithm. It is this new formulation that most explicitly captures the essence of the dynamic programming technique, and it will serve as a general template for the algorithms we develop in later sections.

Designing the Algorithm

The key to the efficient algorithm is really the array \(M\). It encodes the notion that we are using the value of optimal solutions to the subproblems on intervals \(\{1, 2, \ldots, j\}\) for each \(j\), and it uses (6.1) to define the value of \(M[j]\) based on
values that come earlier in the array. Once we have the array $M$, the problem is solved: $M[n]$ contains the value of the optimal solution on the full instance, and $\text{Find-Solution}$ can be used to trace back through $M$ efficiently and return an optimal solution itself.

The point to realize, then, is that we can directly compute the entries in $M$ by an iterative algorithm, rather than using memoized recursion. We just start with $M[0] = 0$ and keep incrementing $j$; each time we need to determine a value $M[j]$, the answer is provided by (6.1). The algorithm looks as follows.

\begin{verbatim}
Iterative-Compute-Opt
    $M[0] = 0$
    For $j = 1, 2, \ldots, n$
        $M[j] = \max(v_j + M[p(j)], M[j - 1])$
    Endfor
\end{verbatim}

\section*{Analyzing the Algorithm}
By exact analogy with the proof of (6.3), we can prove by induction on $j$ that this algorithm writes $\text{OPT}(j)$ in array entry $M[j]$; (6.1) provides the induction step. Also, as before, we can pass the filled-in array $M$ to $\text{Find-Solution}$ to get an optimal solution in addition to the value. Finally, the running time of $\text{Iterative-Compute-Opt}$ is clearly $O(n)$, since it explicitly runs for $n$ iterations and spends constant time in each.

An example of the execution of $\text{Iterative-Compute-Opt}$ is depicted in Figure 6.5. In each iteration, the algorithm fills in one additional entry of the array $M$, by comparing the value of $v_j + M[p(j)]$ to the value of $M[j - 1]$.

\section*{A Basic Outline of Dynamic Programming}
This, then, provides a second efficient algorithm to solve the Weighted Interval Scheduling Problem. The two approaches clearly have a great deal of conceptual overlap, since they both grow from the insight contained in the recurrence (6.1). For the remainder of the chapter, we will develop dynamic programming algorithms using the second type of approach—iterative building up of subproblems—because the algorithms are often simpler to express this way. But in each case that we consider, there is an equivalent way to formulate the algorithm as a memoized recursion.

Most crucially, the bulk of our discussion about the particular problem of selecting intervals can be cast more generally as a rough template for designing dynamic programming algorithms. To set about developing an algorithm based on dynamic programming, one needs a collection of subproblems derived from the original problem that satisfies a few basic properties.
Chapter 6 Dynamic Programming

(i) There are only a polynomial number of subproblems.

(ii) The solution to the original problem can be easily computed from the solutions to the subproblems. (For example, the original problem may actually be one of the subproblems.)

(iii) There is a natural ordering on subproblems from “smallest” to “largest,” together with an easy-to-compute recurrence (as in (6.1) and (6.2)) that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

Naturally, these are informal guidelines. In particular, the notion of “smaller” in part (iii) will depend on the type of recurrence one has.

We will see that it is sometimes easier to start the process of designing such an algorithm by formulating a set of subproblems that looks natural, and then figuring out a recurrence that links them together; but often (as happened in the case of weighted interval scheduling), it can be useful to first define a recurrence by reasoning about the structure of an optimal solution, and then determine which subproblems will be necessary to unwind the recurrence. This chicken-and-egg relationship between subproblems and recurrences is a subtle issue underlying dynamic programming. It’s never clear that a collection of subproblems will be useful until one finds a recurrence linking them together; but it can be difficult to think about recurrences in the absence of the “smaller” subproblems that they build on. In subsequent sections, we will develop further practice in managing this design trade-off.
6.4 Subset Sums and Knapsacks: Adding a Variable

We’re seeing more and more that issues in scheduling provide a rich source of practically motivated algorithmic problems. So far we’ve considered problems in which requests are specified by a given interval of time on a resource, as well as problems in which requests have a duration and a deadline but do not mandate a particular interval during which they need to be done.

In this section, we consider a version of the second type of problem, with durations and deadlines, which is difficult to solve directly using the techniques we’ve seen so far. We will use dynamic programming to solve the problem, but with a twist: the “obvious” set of subproblems will turn out not to be enough, and so we end up creating a richer collection of subproblems. As
6.4 Subset Sums and Knapsacks: Adding a Variable

we will see, this is done by adding a new variable to the recurrence underlying
the dynamic program.

The Problem
In the scheduling problem we consider here, we have a single machine that
can process jobs, and we have a set of requests \( \{1, 2, \ldots, n\} \). We are only
able to use this resource for the period between time 0 and time \( W \), for some
number \( W \). Each request corresponds to a job that requires time \( w_i \) to process.
If our goal is to process jobs so as to keep the machine as busy as possible up
to the “cut-off” \( W \), which jobs should we choose?

More formally, we are given \( n \) items \( \{1, \ldots, n\} \), and each has a given
nonnegative weight \( w_i \) (for \( i = 1, \ldots, n \)). We are also given a bound \( W \). We
would like to select a subset \( S \) of the items so that \( \sum_{i \in S} w_i \leq W \) and, subject
to this restriction, \( \sum_{i \in S} w_i \) is as large as possible. We will call this the Subset
Sum Problem.

This problem is a natural special case of a more general problem called the
Knapsack Problem, where each request \( i \) has both a value \( v_i \) and a weight \( w_i \).
The goal in this more general problem is to select a subset of maximum total
value, subject to the restriction that its total weight not exceed \( W \). Knapsack
problems often show up as subproblems in other, more complex problems. The
name knapsack refers to the problem of filling a knapsack of capacity \( W \) as
full as possible (or packing in as much value as possible), using a subset of the
items \( \{1, \ldots, n\} \). We will use weight or time when referring to the quantities
\( w_i \) and \( W \).

Since this resembles other scheduling problems we’ve seen before, it’s
natural to ask whether a greedy algorithm can find the optimal solution. It
appears that the answer is no—at least, no efficient greedy rule is known that
always constructs an optimal solution. One natural greedy approach to try
would be to sort the items by decreasing weight—or at least to do this for all
items of weight at most \( W \)—and then start selecting items in this order as long
as the total weight remains below \( W \). But if \( W \) is a multiple of 2, and we have
three items with weights \( \{W/2 + 1, W/2, W/2\} \), then we see that this greedy
algorithm will not produce the optimal solution. Alternately, we could sort by
increasing weight and then do the same thing; but this fails on inputs like
\( \{1, W/2, W/2\} \).

The goal of this section is to show how to use dynamic programming to
solve this problem. Recall the main principles of dynamic programming: We
have to come up with a small number of subproblems so that each subproblem
can be solved easily from “smaller” subproblems, and the solution to the
original problem can be obtained easily once we know the solutions to all

For any query, contact us at help@tutorialsduniya.com
the subproblems. The tricky issue here lies in figuring out a good set of subproblems.

**Designing the Algorithm**

**A False Start** One general strategy, which worked for us in the case of Weighted Interval Scheduling, is to consider subproblems involving only the first \( i \) requests. We start by trying this strategy here. We use the notation \( \text{OPT}(i) \), analogously to the notation used before, to denote the best possible solution using a subset of the requests \( \{1, \ldots, i\} \). The key to our method for the Weighted Interval Scheduling Problem was to concentrate on an optimal solution \( \emptyset \) to our problem and consider two cases, depending on whether or not the last request \( n \) is accepted or rejected by this optimum solution. Just as in that case, we have the first part, which follows immediately from the definition of \( \text{OPT}(i) \).

- If \( n \notin \emptyset \), then \( \text{OPT}(n) = \text{OPT}(n - 1) \).

Next we have to consider the case in which \( n \in \emptyset \). What we’d like here is a simple recursion, which tells us the best possible value we can get for solutions that contain the last request \( n \). For Weighted Interval Scheduling this was easy, as we could simply delete each request that conflicted with request \( n \). In the current problem, this is not so simple. Accepting request \( n \) does not immediately imply that we have to reject any other request. Instead, it means that for the subset of requests \( S \subseteq \{1, \ldots, n - 1\} \) that we will accept, we have less available weight left: a weight of \( w_n \) is used on the accepted request \( n \), and we only have \( W - w_n \) weight left for the set \( S \) of remaining requests that we accept. See Figure 6.10.

**A Better Solution** This suggests that we need more subproblems: To find out the value for \( \text{OPT}(n) \) we not only need the value of \( \text{OPT}(n - 1) \), but we also need to know the best solution we can get using a subset of the first \( n - 1 \) items and total allowed weight \( W - w_n \). We are therefore going to use many more subproblems: one for each initial set \( \{1, \ldots, i\} \) of the items, and each possible

![Figure 6.10](https://example.com/figure610.png)

**Figure 6.10** After item \( n \) is included in the solution, a weight of \( w_n \) is used up and there is \( W - w_n \) available weight left.
value for the remaining available weight $w$. Assume that $W$ is an integer, and all requests $i = 1, \ldots, n$ have integer weights $w_i$. We will have a subproblem for each $i = 0, 1, \ldots, n$ and each integer $0 \leq w \leq W$. We will use $\text{OPT}(i, w)$ to denote the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ with maximum allowed weight $w$, that is,

$$\text{OPT}(i, w) = \max_S \sum_{j \in S} w_j,$$

where the maximum is over subsets $S \subseteq \{1, \ldots, i\}$ that satisfy $\sum_{j \in S} w_j \leq w$. Using this new set of subproblems, we will be able to express the value $\text{OPT}(i, w)$ as a simple expression in terms of values from smaller problems. Moreover, $\text{OPT}(n, W)$ is the quantity we’re looking for in the end. As before, let $\mathcal{O}$ denote an optimum solution for the original problem.

- If $n \not\in \mathcal{O}$, then $\text{OPT}(n, W) = \text{OPT}(n - 1, W)$, since we can simply ignore item $n$.
- If $n \in \mathcal{O}$, then $\text{OPT}(n, W) = w_n + \text{OPT}(n - 1, W - w_n)$, since we now seek to use the remaining capacity of $W - w_n$ in an optimal way across items $1, 2, \ldots, n - 1$.

When the $n^{\text{th}}$ item is too big, that is, $W < w_n$, then we must have $\text{OPT}(n, W) = \text{OPT}(n - 1, W)$. Otherwise, we get the optimum solution allowing all $n$ requests by taking the better of these two options. Using the same line of argument for the subproblem for items $\{1, \ldots, i\}$, and maximum allowed weight $w$, gives us the following recurrence.

\begin{equation}
\text{OPT}(i, w) = \begin{cases} 
\text{OPT}(i - 1, w), & \text{if } w < w_i \\
\max(\text{OPT}(i - 1, w), w_i + \text{OPT}(i - 1, w - w_i)), & \text{otherwise}
\end{cases}
\end{equation}

As before, we want to design an algorithm that builds up a table of all $\text{OPT}(i, w)$ values while computing each of them at most once.

---

**Subset-Sum($n, W$)**

Array $M[0 \ldots n, 0 \ldots W]$

Initialize $M[0, w] = 0$ for each $w = 0, 1, \ldots, W$

For $i = 1, 2, \ldots, n$
    For $w = 0, \ldots, W$
        Use the recurrence (6.8) to compute $M[i, w]$
     Endfor
    Endfor
Return $M[n, W]$

---

For any query, contact us at help@tutorialsduniya.com
Chapter 6  Dynamic Programming

Figure 6.11  The two-dimensional table of $\text{OPT}$ values. The leftmost column and bottom row is always 0. The entry for $\text{OPT}(i, w)$ is computed from the two other entries $\text{OPT}(i - 1, w)$ and $\text{OPT}(i - 1, w - w_i)$, as indicated by the arrows.

Using (6.8) one can immediately prove by induction that the returned value $M[n, W]$ is the optimum solution value for the requests $1, \ldots, n$ and available weight $W$.

Analyzing the Algorithm

Recall the tabular picture we considered in Figure 6.5, associated with weighted interval scheduling, where we also showed the way in which the array $M$ for that algorithm was iteratively filled in. For the algorithm we’ve just designed, we can use a similar representation, but we need a two-dimensional table, reflecting the two-dimensional array of subproblems that is being built up. Figure 6.11 shows the building up of subproblems in this case: the value $M[i, w]$ is computed from the two other values $M[i - 1, w]$ and $M[i - 1, w - w_i]$.

As an example of this algorithm executing, consider an instance with weight limit $W = 6$, and $n = 3$ items of sizes $w_1 = w_2 = 2$ and $w_3 = 3$. We find that the optimal value $\text{OPT}(3, 6) = 5$ (which we get by using the third item and one of the first two items). Figure 6.12 illustrates the way the algorithm fills in the two-dimensional table of $\text{OPT}$ values row by row.

Next we will worry about the running time of this algorithm. As before in the case of weighted interval scheduling, we are building up a table of solutions $M$, and we compute each of the values $M[i, w]$ in $O(1)$ time using the previous values. Thus the running time is proportional to the number of entries in the table.
6.4 Subset Sums and Knapsacks: Adding a Variable

Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

![Initial values](image1)

![Filling in values for $i = 1$](image2)

![Filling in values for $i = 2$](image3)

![Filling in values for $i = 3$](image4)

Figure 6.12 The iterations of the algorithm on a sample instance of the Subset Sum Problem.

(6.9) The Subset-Sum$(n,W)$ Algorithm correctly computes the optimal value of the problem, and runs in $O(nW)$ time.

Note that this method is not as efficient as our dynamic program for the Weighted Interval Scheduling Problem. Indeed, its running time is not a polynomial function of $n$; rather, it is a polynomial function of $n$ and $W$, the largest integer involved in defining the problem. We call such algorithms pseudo-polynomial. Pseudo-polynomial algorithms can be reasonably efficient when the numbers $\{w_i\}$ involved in the input are reasonably small; however, they become less practical as these numbers grow large.

To recover an optimal set $S$ of items, we can trace back through the array $M$ by a procedure similar to those we developed in the previous sections.

(6.10) Given a table $M$ of the optimal values of the subproblems, the optimal set $S$ can be found in $O(n)$ time.

Extension: The Knapsack Problem

The Knapsack Problem is a bit more complex than the scheduling problem we discussed earlier. Consider a situation in which each item $i$ has a nonnegative weight $w_i$ as before, and also a distinct value $v_i$. Our goal is now to find a
Chapter 6 Dynamic Programming

subset $S$ of maximum value $\sum_{i \in S} v_i$, subject to the restriction that the total weight of the set should not exceed $W$: $\sum_{i \in S} w_i \leq W$.

It is not hard to extend our dynamic programming algorithm to this more general problem. We use the analogous set of subproblems, $OPT(i, w)$, to denote the value of the optimal solution using a subset of the items $\{1, \ldots, i\}$ and maximum available weight $w$. We consider an optimal solution $O$, and identify two cases depending on whether or not $n \in O$.

- If $n \not\in O$, then $OPT(n, W) = OPT(n - 1, W)$.
- If $n \in O$, then $OPT(n, W) = v_n + OPT(n - 1, W - w_n)$.

Using this line of argument for the subproblems implies the following analogue of (6.8).

(6.11) If $w < w_i$ then $OPT(i, w) = OPT(i - 1, w)$. Otherwise

$$OPT(i, w) = \max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)).$$

Using this recurrence, we can write down a completely analogous dynamic programming algorithm, and this implies the following fact.

(6.12) The Knapsack Problem can be solved in $O(nW)$ time.
Solved Exercises

Solved Exercise 1

Suppose you are managing the construction of billboards on the Stephen Daedalus Memorial Highway, a heavily traveled stretch of road that runs west-east for $M$ miles. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$, each in the interval $[0, M]$ (specifying their position along the highway, measured in miles from its western end). If you place a billboard at location $x_i$, you receive a revenue of $r_i > 0$.

Regulations imposed by the county’s Highway Department require that no two of the billboards be within less than or equal to 5 miles of each other. You’d like to place billboards at a subset of the sites so as to maximize your total revenue, subject to this restriction.

Example. Suppose $M = 20$, $n = 4$,

$$\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\},$$

and

$$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}.$$  

Then the optimal solution would be to place billboards at $x_1$ and $x_3$, for a total revenue of 10.

Give an algorithm that takes an instance of this problem as input and returns the maximum total revenue that can be obtained from any valid subset of sites. The running time of the algorithm should be polynomial in $n$.

Solution  We can naturally apply dynamic programming to this problem if we reason as follows. Consider an optimal solution for a given input instance; in this solution, we either place a billboard at site $x_n$ or not. If we don’t, the optimal solution on sites $x_1, \ldots, x_n$ is really the same as the optimal solution
on sites $x_1, \ldots, x_{n-1}$; if we do, then we should eliminate $x_n$ and all other sites that are within 5 miles of it, and find an optimal solution on what’s left. The same reasoning applies when we’re looking at the problem defined by just the first $j$ sites, $x_1, \ldots, x_j$: we either include $x_j$ in the optimal solution or we don’t, with the same consequences.

Let’s define some notation to help express this. For a site $x_j$, we let $e(j)$ denote the easternmost site $x_i$ that is more than 5 miles from $x_j$. Since sites are numbered west to east, this means that the sites $x_1, x_2, \ldots, x_{e(j)}$ are still valid options once we’ve chosen to place a billboard at $x_j$, but the sites $x_{e(j)+1}, \ldots, x_{j-1}$ are not.

Now, our reasoning above justifies the following recurrence. If we let $OPT(j)$ denote the revenue from the optimal subset of sites among $x_1, \ldots, x_j$, then we have

$$OPT(j) = \max(r_j + OPT(e(j)), OPT(j - 1)).$$

We now have most of the ingredients we need for a dynamic programming algorithm. First, we have a set of $n$ subproblems, consisting of the first $j$ sites for $j = 0, 1, 2, \ldots, n$. Second, we have a recurrence that lets us build up the solutions to subproblems, given by $OPT(j) = \max(r_j + OPT(e(j)), OPT(j - 1))$.

To turn this into an algorithm, we just need to define an array $M$ that will store the $OPT$ values and throw a loop around the recurrence that builds up the values $M[j]$ in order of increasing $j$.

---

Initialize $M[0] = 0$ and $M[1] = r_1$
For $j = 2, 3, \ldots, n$:
  Compute $M[j]$ using the recurrence
Endfor
Return $M[n]$

---

As with all the dynamic programming algorithms we’ve seen in this chapter, an optimal set of billboards can be found by tracing back through the values in array $M$.

Given the values $e(j)$ for all $j$, the running time of the algorithm is $O(n)$, since each iteration of the loop takes constant time. We can also compute all $e(j)$ values in $O(n)$ time as follows. For each site location $x_i$, we define $x'_i = x_i - 5$. We then merge the sorted list $x_1, \ldots, x_n$ with the sorted list $x'_1, \ldots, x'_n$ in linear time, as we saw how to do in Chapter 2. We now scan through this merged list; when we get to the entry $x'_j$, we know that anything from this point onward to $x_j$ cannot be chosen together with $x_j$ (since it’s within 5 miles), and so we
simply define $e(j)$ to be the largest value of $i$ for which we've seen $x_i$ in our scan.

Here's a final observation on this problem. Clearly, the solution looks very much like that of the Weighted Interval Scheduling Problem, and there's a fundamental reason for that. In fact, our billboard placement problem can be directly encoded as an instance of Weighted Interval Scheduling, as follows. Suppose that for each site $x_i$, we define an interval with endpoints $[x_i - 5, x_i]$ and weight $r_i$. Then, given any nonoverlapping set of intervals, the corresponding set of sites has the property that no two lie within 5 miles of each other. Conversely, given any such set of sites (no two within 5 miles), the intervals associated with them will be nonoverlapping. Thus the collections of nonoverlapping intervals correspond precisely to the set of valid billboard placements, and so dropping the set of intervals we've just defined (with their weights) into an algorithm for Weighted Interval Scheduling will yield the desired solution.

Solved Exercise 2

Through some friends of friends, you end up on a consulting visit to the cutting-edge biotech firm Clones ‘R’ Us (CRU). At first you're not sure how your algorithmic background will be of any help to them, but you soon find yourself called upon to help two identical-looking software engineers tackle a perplexing problem.

The problem they are currently working on is based on the concatenation of sequences of genetic material. If $X$ and $Y$ are each strings over a fixed alphabet $S$, then $XY$ denotes the string obtained by concatenating them—writing $X$ followed by $Y$. CRU has identified a target sequence $A$ of genetic material, consisting of $m$ symbols, and they want to produce a sequence that is as similar to $A$ as possible. For this purpose, they have a library $L$ consisting of $k$ (shorter) sequences, each of length at most $n$. They can cheaply produce any sequence consisting of copies of the strings in $L$ concatenated together (with repetitions allowed).

Thus we say that a concatenation over $L$ is any sequence of the form $B_1B_2\cdots B_\ell$, where each $B_i$ belongs the set $L$. (Again, repetitions are allowed, so $B_i$ and $B_j$ could be the same string in $L$, for different values of $i$ and $j$.) The problem is to find a concatenation over $\{B_i\}$ for which the sequence alignment cost is as small as possible. (For the purpose of computing the sequence alignment cost, you may assume that you are given a gap cost $\delta$ and a mismatch cost $\alpha_{pq}$ for each pair $p, q \in S$.)

Give a polynomial-time algorithm for this problem.
**Solution** This problem is vaguely reminiscent of Segmented Least Squares: we have a long sequence of “data” (the string $A$) that we want to “fit” with shorter segments (the strings in $L$).

If we wanted to pursue this analogy, we could search for a solution as follows. Let $B = B_1B_2 \cdots B_\ell$ denote a concatenation over $L$ that aligns as well as possible with the given string $A$. (That is, $B$ is an optimal solution to the input instance.) Consider an optimal alignment $M$ of $A$ with $B$, let $t$ be the first position in $A$ that is matched with some symbol in $B_\ell$, and let $A_\ell$ denote the substring of $A$ from position $t$ to the end. (See Figure 6.27 for an illustration of this with $\ell = 3$.) Now, the point is that in this optimal alignment $M$, the substring $A_\ell$ is optimally aligned with $B_\ell$; indeed, if there were a way to better align $A_\ell$ with $B_\ell$, we could substitute it for the portion of $M$ that aligns $A_\ell$ with $B_\ell$ and obtain a better overall alignment of $A$ with $B$.

This tells us that we can look at the optimal solution as follows. There’s some final piece of $A_\ell$ that is aligned with one of the strings in $L$, and for this piece all we’re doing is finding the string in $L$ that aligns with it as well as possible. Having found this optimal alignment for $A_\ell$, we can break it off and continue to find the optimal solution for the remainder of $A$.

Thinking about the problem this way doesn’t tell us exactly how to proceed—we don’t know how long $A_\ell$ is supposed to be, or which string in $L$ it should be aligned with. But this is the kind of thing we can search over in a dynamic programming algorithm. Essentially, we’re in about the same spot we were in with the Segmented Least Squares Problem: there we knew that we had to break off some final subsequence of the input points, fit them as well as possible with one line, and then iterate on the remaining input points.

So let’s set up things to make the search for $A_\ell$ possible. First, let $A[x:y]$ denote the substring of $A$ consisting of its symbols from position $x$ to position $y$, inclusive. Let $c(x, y)$ denote the cost of the optimal alignment of $A[x:y]$ with any string in $L$. (That is, we search over each string in $L$ and find the one that

![Figure 6.27](image-url) In the optimal concatenation of strings to align with $A$, there is a final string ($B_3$ in the figure) that aligns with a substring of $A$ ($A_3$ in the figure) that extends from some position $t$ to the end.
aligns best with \( A[x:y] \). Let \( \text{OPT}(j) \) denote the alignment cost of the optimal solution on the string \( A[1:j] \).

The argument above says that an optimal solution on \( A[1:j] \) consists of identifying a final “segment boundary” \( t < j \), finding the optimal alignment of \( A[t:j] \) with a single string in \( \mathcal{L} \), and iterating on \( A[1:t−1] \). The cost of this alignment of \( A[t:j] \) is just \( c(t,j) \), and the cost of aligning with what’s left is just \( \text{OPT}(t−1) \). This suggests that our subproblems fit together very nicely, and it justifies the following recurrence.

\[
(6.37) \quad \text{OPT}(j) = \min_{t<j} c(t,j) + \text{OPT}(t−1) \quad \text{for } j \geq 1, \text{ and } \text{OPT}(0) = 0.
\]

The full algorithm consists of first computing the quantities \( c(t,j) \), for \( t < j \), and then building up the values \( \text{OPT}(j) \) in order of increasing \( j \). We hold these values in an array \( M \).

---

Set \( M[0] = 0 \)

For all pairs \( 1 \leq t \leq j \leq m \)

- Compute the cost \( c(t,j) \) as follows:
  - For each string \( B \in \mathcal{L} \)
    - Compute the optimal alignment of \( B \) with \( A[t:j] \)
    - Endfor
  - Choose the \( B \) that achieves the best alignment, and use this alignment cost as \( c(t,j) \)
    - Endfor
  - For \( j = 1, 2, \ldots, n \)
    - Use the recurrence (6.37) to compute \( M[j] \)
    - Endfor
  - Return \( M[n] \)

As usual, we can get a concatenation that achieves it by tracing back over the array of \( \text{OPT} \) values.

Let’s consider the running time of this algorithm. First, there are \( O(m^2) \) values \( c(t,j) \) that need to be computed. For each, we try each string of the \( k \) strings \( B \in \mathcal{L} \), and compute the optimal alignment of \( B \) with \( A[t:j] \) in time \( O(n(j−t)) = O(mn) \). Thus the total time to compute all \( c(t,j) \) values is \( O(km^3n) \).

This dominates the time to compute all \( \text{OPT} \) values: Computing \( \text{OPT}(j) \) uses the recurrence in (6.37), and this takes \( O(m) \) time to compute the minimum. Summing this over all choices of \( j = 1, 2, \ldots, m \), we get \( O(m^2) \) time for this portion of the algorithm.
Chapter 6 Dynamic Programming

Exercises

1. Let \( G = (V, E) \) be an undirected graph with \( n \) nodes. Recall that a subset of the nodes is called an independent set if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we’ll see that it can be done efficiently if the graph is “simple” enough.

Call a graph \( G = (V, E) \) a path if its nodes can be written as \( v_1, v_2, \ldots, v_n \), with an edge between \( v_i \) and \( v_j \) if and only if the numbers \( i \) and \( j \) differ by exactly 1. With each node \( v_i \), we associate a positive integer weight \( w_i \).

Consider, for example, the five-node path drawn in Figure 6.28. The weights are the numbers drawn inside the nodes.

The goal in this question is to solve the following problem:

Find an independent set in a path \( G \) whose total weight is as large as possible.

(a) Give an example to show that the following algorithm does not always find an independent set of maximum total weight.

The "heaviest-first" greedy algorithm

Start with \( S \) equal to the empty set
While some node remains in \( G \)
    Pick a node \( v_i \) of maximum weight
    Add \( v_i \) to \( S \)
    Delete \( v_i \) and its neighbors from \( G \)
Endwhile
Return \( S \)

(b) Give an example to show that the following algorithm also does not always find an independent set of maximum total weight.

Let \( S_1 \) be the set of all \( v_i \) where \( i \) is an odd number
Let \( S_2 \) be the set of all \( v_i \) where \( i \) is an even number
(Note that \( S_1 \) and \( S_2 \) are both independent sets)
Determine which of \( S_1 \) or \( S_2 \) has greater total weight, and return this one

Figure 6.28 A path with weights on the nodes. The maximum weight of an independent set is 14.
(c) Give an algorithm that takes an \( n \)-node path \( G \) with weights and returns an independent set of maximum total weight. The running time should be polynomial in \( n \), independent of the values of the weights.

2. Suppose you’re managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are low-stress (e.g., setting up a Web site for a class at the local elementary school) and those that are high-stress (e.g., protecting the nation’s most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week \( i \), then you get a revenue of \( \ell_i > 0 \) dollars; if you select a high-stress job, you get a revenue of \( h_i > 0 \) dollars. The catch, however, is that in order for the team to take on a high-stress job in week \( i \), it’s required that they do no job (of either type) in week \( i - 1 \); they need a full week of prep time to get ready for the crushing stress level. On the other hand, it’s okay for them to take a low-stress job in week \( i \) even if they have done a job (of either type) in week \( i - 1 \).

So, given a sequence of \( n \) weeks, a plan is specified by a choice of “low-stress,” “high-stress,” or “none” for each of the \( n \) weeks, with the property that if “high-stress” is chosen for week \( i > 1 \), then “none” has to be chosen for week \( i - 1 \). (It’s okay to choose a high-stress job in week 1.) The value of the plan is determined in the natural way: for each \( i \), you add \( \ell_i \) to the value if you choose “low-stress” in week \( i \), and you add \( h_i \) to the value if you choose “high-stress” in week \( i \). (You add 0 if you choose “none” in week \( i \).)

**The problem.** Given sets of values \( \ell_1, \ell_2, \ldots, \ell_n \) and \( h_1, h_2, \ldots, h_n \), find a plan of maximum value. (Such a plan will be called optimal.)

**Example.** Suppose \( n = 4 \), and the values of \( \ell_i \) and \( h_i \) are given by the following table. Then the plan of maximum value would be to choose “none” in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be \( 0 + 50 + 10 + 10 = 70 \).

<table>
<thead>
<tr>
<th></th>
<th>Week 1</th>
<th>Week 2</th>
<th>Week 3</th>
<th>Week 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell )</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( h )</td>
<td>5</td>
<td>50</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 6 Dynamic Programming

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

For iterations $i = 1$ to $n$
  If $h_{i+1} > \ell_i + \ell_{i+1}$ then
    Output "Choose no job in week $i"
    Output "Choose a high-stress job in week $i+1"
    Continue with iteration $i+2$
  Else
    Output "Choose a low-stress job in week $i"
    Continue with iteration $i+1$
  Endif
End

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an efficient algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$ and returns the value of an optimal plan.

3. Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that $G$ is an ordered graph if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form $(v_i, v_j)$ with $i < j$.

(ii) Each node except $v_n$ has at least one edge leaving it. That is, for every node $v_i$, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form $(v_i, v_j)$.

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

Given an ordered graph $G$, find the length of the longest path that begins at $v_1$ and ends at $v_n$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

Set $w = v_1$
Set $L = 0$
Figure 6.29 The correct answer for this ordered graph is 3: The longest path from \( v_1 \) to \( v_n \) uses the three edges \((v_1, v_2), (v_2, v_4), \) and \((v_4, v_5)\).

While there is an edge out of the node \( w \)

Choose the edge \((w, v_j)\)

for which \( j \) is as small as possible

Set \( w = v_j \)

Increase \( L \) by 1

end while

Return \( L \) as the length of the longest path

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph \( G \) and returns the length of the longest path that begins at \( v_1 \) and ends at \( v_n \). (Again, the length of a path is the number of edges in the path.)

4. Suppose you’re running a lightweight consulting business—just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month \( i \), you’ll incur an operating cost of \( N_i \) if you run the business out of NY; you’ll incur an operating cost of \( S_i \) if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month \( i \), and then out of the other city in month \( i + 1 \), then you incur a fixed moving cost of \( M \) to switch base offices.

Given a sequence of \( n \) months, a plan is a sequence of \( n \) locations—each one equal to either NY or SF—such that the \( i \)th location indicates the city in which you will be based in the \( i \)th month. The cost of a plan is the sum of the operating costs for each of the \( n \) months, plus a moving cost of \( M \) for each time you switch cities. The plan can begin in either city.
Chapter 6 Dynamic Programming

The problem. Given a value for the moving cost $M$, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called optimal.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

<table>
<thead>
<tr>
<th></th>
<th>Month 1</th>
<th>Month 2</th>
<th>Month 3</th>
<th>Month 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NY</td>
<td>1</td>
<td>3</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>SF</td>
<td>50</td>
<td>20</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Then the plan of minimum cost would be the sequence of locations $[NY, NY, SF, SF]$, with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

For $i = 1$ to $n$

  If $N_i < S_i$ then
  
  Output "NY in Month $i"

  Else
  
  Output "SF in Month $i"

End

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an example of an instance in which every optimal plan must move (i.e., change locations) at least three times.

  Provide a brief explanation, saying why your example has this property.

(c) Give an efficient algorithm that takes values for $n$, $M$, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the cost of an optimal plan.

5. As some of you know well, and others of you may be interested to learn, a number of languages (including Chinese and Japanese) are written without spaces between the words. Consequently, software that works with text written in these languages must address the word segmentation problem—inferring likely boundaries between consecutive words in the
text. If English were written without spaces, the analogous problem would
consist of taking a string like “meetateight” and deciding that the best
segmentation is “meet at eight” (and not “me et at eight,” or “meet ate
ight,” or any of a huge number of even less plausible alternatives). How
could we automate this process?

A simple approach that is at least reasonably effective is to find a
segmentation that simply maximizes the cumulative “quality” of its indi-
vidual constituent words. Thus, suppose you are given a black box that,
for any string of letters \( x = x_1x_2 \cdots x_k \), will return a number \( \text{quality}(x) \). This
number can be either positive or negative; larger numbers correspond to
more plausible English words. (So \( \text{quality(“me”)} \) would be positive, while
\( \text{quality(“ght”)} \) would be negative.)

Given a long string of letters \( y = y_1y_2 \cdots y_n \), a segmentation of \( y \) is a
partition of its letters into contiguous blocks of letters; each block corre-
sponds to a word in the segmentation. The total quality of a segmentation
is determined by adding up the qualities of each of its blocks. (So we’d
get the right answer above provided that \( \text{quality(“meet”) + quality(“at”) + \text{quality(“eight”) would be positive.} \)

Give an efficient algorithm that takes a string \( y \) and computes a
segmentation of maximum total quality. (You can treat a single call to
the black box computing \( \text{quality}(x) \) as a single computational step.)

(A final note, not necessary for solving the problem: To achieve better
performance, word segmentation software in practice works with a more
complex formulation of the problem—for example, incorporating the
notion that solutions should not only be reasonable at the word level, but
also form coherent phrases and sentences. If we consider the example
“theyouthevent,” there are at least three valid ways to segment this
into common English words, but one constitutes a much more coherent
phrase than the other two. If we think of this in the terminology of formal
languages, this broader problem is like searching for a segmentation
that also can be parsed well according to a grammar for the underlying
language. But even with these additional criteria and constraints, dynamic
programming approaches lie at the heart of a number of successful
segmentation systems.)

6. In a word processor, the goal of “pretty-printing” is to take text with a
ragged right margin, like this,

Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,  
and nothing particular to interest me on shore,  
I thought I would sail about a little  
and see the watery part of the world.

and turn it into text whose right margin is as “even” as possible, like this.

Call me Ishmael. Some years ago, never  
mind how long precisely, having little  
or no money in my purse, and nothing  
particular to interest me on shore, I  
thought I would sail about a little  
and see the watery part of the world.

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be “even.” So suppose our text consists of a sequence of words, \( W = \{ w_1, w_2, \ldots, w_n \} \), where \( w_i \) consists of \( c_i \) characters. We have a maximum line length of \( L \). We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A formatting of \( W \) consists of a partition of the words in \( W \) into lines. In the words assigned to a single line, there should be a space after each word except the last; and so if \( w_j, w_{j+1}, \ldots, w_k \) are assigned to one line, then we should have

\[
\left[ \sum_{i=j}^{k-1} (c_i + 1) \right] + c_k \leq L.
\]

We will call an assignment of words to a line valid if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the slack of the line—that is, the number of spaces left at the right margin.

Give an efficient algorithm to find a partition of a set of words \( W \) into valid lines, so that the sum of the squares of the slacks of all lines (including the last line) is minimized.

7. As a solved exercise in Chapter 5, we gave an algorithm with \( O(n \log n) \) running time for the following problem. We’re looking at the price of a given stock over \( n \) consecutive days, numbered \( i = 1, 2, \ldots, n \). For each day \( i \), we have a price \( p(i) \) per share for the stock on that day. (We’ll assume for simplicity that the price was fixed during each day.) We’d like to know: How should we choose a day \( i \) on which to buy the stock and a later day \( j > i \) on which to sell it, if we want to maximize the profit per

For any query, contact us at help@tutorialsduniya.com
Exercises

share, \( p(j) - p(i) \)? (If there is no way to make money during the \( n \) days, we should conclude this instead.)

In the solved exercise, we showed how to find the optimal pair of days \( i \) and \( j \) in time \( O(n \log n) \). But, in fact, it’s possible to do better than this. Show how to find the optimal numbers \( i \) and \( j \) in time \( O(n) \).

8. The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots.

Here’s what one of these robot attacks looks like.

• A swarm of robots arrives over the course of \( n \) seconds; in the \( i \)th second, \( x_i \) robots arrive. Based on remote sensing data, you know this sequence \( x_1, x_2, \ldots, x_n \) in advance.

• You have at your disposal an *electromagnetic pulse* (EMP), which can destroy some of the robots as they arrive; the EMP’s power depends on how long it’s been allowed to charge up. To make this precise, there is a function \( f(\cdot) \) so that if \( j \) seconds have passed since the EMP was last used, then it is capable of destroying up to \( f(j) \) robots.

• So specifically, if it is used in the \( k \)th second, and it has been \( j \) seconds since it was previously used, then it will destroy \( \min(x_k, f(j)) \) robots. (After this use, it will be completely drained.)

• We will also assume that the EMP starts off completely drained, so if it is used for the first time in the \( j \)th second, then it is capable of destroying up to \( f(j) \) robots.

**The problem.** Given the data on robot arrivals \( x_1, x_2, \ldots, x_n \), and given the recharging function \( f(\cdot) \), choose the points in time at which you’re going to activate the EMP so as to destroy as many robots as possible.

**Example.** Suppose \( n = 4 \), and the values of \( x_i \) and \( f(i) \) are given by the following table.

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>( f(i) )</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

The best solution would be to activate the EMP in the 3rd and the 4th seconds. In the 3rd second, the EMP has gotten to charge for 3 seconds, and so it destroys \( \min(10, 4) = 4 \) robots; in the 4th second, the EMP has only gotten to charge for 1 second since its last use, and it destroys \( \min(1, 1) = 1 \) robot. This is a total of 5.
Chapter 6 Dynamic Programming

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

\[
\text{Schedule-EMP}(x_1, \ldots, x_n) \\
\text{Let } j \text{ be the smallest number for which } f(j) \geq x_n \\
\text{(If no such } j \text{ exists then set } j = n) \\
\text{Activate the EMP in the } n^{th} \text{ second} \\
\text{If } n - j \geq 1 \text{ then} \\
\text{Continue recursively on the input } x_1, \ldots, x_{n-j} \\
\text{(i.e., invoke Schedule-EMP}(x_1, \ldots, x_{n-j}))
\]

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes the data on robot arrivals \( x_1, x_2, \ldots, x_n \), and the recharging function \( f(\cdot) \), and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

9. You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of \( n \) days, you're presented with a quantity of data; on day \( i \), you're presented with \( x_i \) terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process \( s_1 \) terabytes, on the second day after a reboot, you can process \( s_2 \) terabytes, and so on, up to \( s_n \); we assume \( s_1 > s_2 > s_3 > \cdots > s_n > 0 \). (Of course, on day \( i \) you can only process up to \( x_i \) terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data \( x_1, x_2, \ldots, x_n \) for the next \( n \) days, and given the profile of your system as expressed by \( s_1, s_2, \ldots, s_n \) (and starting from a freshly rebooted system on day 1), choose
the days on which you're going to reboot so as to maximize the total amount of data you process.

**Example.** Suppose \( n = 4 \), and the values of \( x_i \) and \( s_i \) are given by the following table.

<table>
<thead>
<tr>
<th></th>
<th>Day 1</th>
<th>Day 2</th>
<th>Day 3</th>
<th>Day 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>( s )</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn’t reboot at all, you’d process \( 8 + 1 + 2 + 1 = 12 \); and other rebooting strategies give you less than 19 as well.)

(a) Give an example of an instance with the following properties.

- There is a “surplus” of data in the sense that \( x_i > s_i \) for every \( i \).
- The optimal solution reboots the system at least twice.

In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

(b) Give an efficient algorithm that takes values for \( x_1, x_2, \ldots, x_n \) and \( s_1, s_2, \ldots, s_n \) and returns the total number of terabytes processed by an optimal solution.

10. You're trying to run a large computing job in which you need to simulate a physical system for as many discrete steps as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next \( n \) minutes, you have a “profile” of how much processing power is available on each machine. In minute \( i \), you would be able to run \( a_i > 0 \) steps of the simulation if your job is on machine A, and \( b_i > 0 \) steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of \( n \) minutes, a plan is specified by a choice of A, B, or “move” for each minute, with the property that choices A and
Chapter 6  Dynamic Programming

$B$ cannot appear in consecutive minutes. For example, if your job is on machine $A$ in minute $i$, and you want to switch to machine $B$, then your choice for minute $i + 1$ must be move, and then your choice for minute $i + 2$ can be $B$. The value of a plan is the total number of steps that you manage to execute over the $n$ minutes: so it’s the sum of $a_i$ over all minutes in which the job is on $A$, plus the sum of $b_i$ over all minutes in which the job is on $B$.

**The problem.** Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal.*) Note that your plan can start with either of the machines $A$ or $B$ in minute 1.

**Example.** Suppose $n = 4$, and the values of $a_i$ and $b_i$ are given by the following table.

<table>
<thead>
<tr>
<th>Minute 1</th>
<th>Minute 2</th>
<th>Minute 3</th>
<th>Minute 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Then the plan of maximum value would be to choose $A$ for minute 1, then *move* for minute 2, and then $B$ for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

---

In minute 1, choose the machine achieving the larger of $a_1$, $b_1$
Set $i = 2$
While $i \leq n$
    What was the choice in minute $i - 1$?
    If $A$:
        If $b_{i+1} > a_i + a_{i+1}$ then
            Choose *move* in minute $i$ and $B$ in minute $i + 1$
            Proceed to iteration $i + 2$
        Else
            Choose $A$ in minute $i$
            Proceed to iteration $i + 1$
        Endif
    Else
        Choose $B$ in minute $i$
        Proceed to iteration $i + 1$
    Endif
EndWhile

---

For any query, contact us at help@tutorialsduniya.com
In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for \( a_1, a_2, \ldots, a_n \) and \( b_1, b_2, \ldots, b_n \) and returns the value of an optimal plan.

11. Suppose you're consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next \( n \) weeks, they have a projected supply \( s_i \) of equipment (measured in pounds), which has to be shipped by an air freight carrier.

Each week's supply can be carried by one of two air freight companies, A or B.
- Company A charges a fixed rate \( r \) per pound (so it costs \( r \cdot s_i \) to ship a week's supply \( s_i \)).
- Company B makes contracts for a fixed amount \( c \) per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A schedule, for the PC company, is a choice of air freight company (A or B) for each of the \( n \) weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The cost of the schedule is the total amount paid to company A and B, according to the description above.

Give a polynomial-time algorithm that takes a sequence of supply values \( s_1, s_2, \ldots, s_n \) and returns a schedule of minimum cost.

**Example.** Suppose \( r = 1 \), \( c = 10 \), and the sequence of values is

\[
11, 9, 9, 12, 12, 12, 9, 9, 11.
\]

Then the optimal schedule would be to choose company A for the first three weeks, then company B for a block of four consecutive weeks, and then company A for the final three weeks.

12. Suppose we want to replicate a file over a collection of \( n \) servers, labeled \( S_1, S_2, \ldots, S_n \). To place a copy of the file at server \( S_i \) results in a placement cost of \( c_i \), for an integer \( c_i > 0 \).

Now, if a user requests the file from server \( S_i \) and no copy of the file is present at \( S_i \), then the servers \( S_{i+1}, S_{i+2}, S_{i+3}, \ldots \) are searched in order until a copy of the file is finally found, say at server \( S_j \), where \( j > i \). This results in an access cost of \( j - i \). (Note that the lower-indexed servers \( S_{i-1}, S_{i-2}, \ldots \) are not consulted in this search.) The access cost is 0 if \( S_i \) holds a copy of the file. We will require that a copy of the file be placed at server \( S_n \), so that all such searches will terminate, at the latest, at \( S_n \).
Chapter 6 Dynamic Programming

We'd like to place copies of the files at the servers so as to minimize the sum of placement and access costs. Formally, we say that a configuration is a choice, for each server \( S_i \) with \( i = 1, 2, \ldots, n - 1 \), of whether to place a copy of the file at \( S_i \) or not. (Recall that a copy is always placed at \( S_n \).) The total cost of a configuration is the sum of all placement costs for servers with a copy of the file, plus the sum of all access costs associated with all \( n \) servers.

Give a polynomial-time algorithm to find a configuration of minimum total cost.

13. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in \( n \) different companies. For each pair \( i \neq j \), they maintain a trade ratio \( r_{ij} \), meaning that one share of \( i \) trades for \( r_{ij} \) shares of \( j \). Here we allow the rate \( r \) to be fractional; that is, \( r_{ij} = \frac{2}{3} \) means that you can trade three shares of \( i \) to get two shares of \( j \).

A trading cycle for a sequence of shares \( i_1, i_2, \ldots, i_k \) consists of successively trading shares in company \( i_1 \) for shares in company \( i_2 \), then shares in company \( i_2 \) for shares \( i_3 \), and so on, finally trading shares in \( i_k \) back to shares in company \( i_1 \). After such a sequence of trades, one ends up with shares in the same company \( i_1 \) that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an opportunity cycle, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

Give a polynomial-time algorithm that finds such an opportunity cycle, if one exists.

14. A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices \( x \) and \( y \) are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge \((x, y)\) might disappear as \( x \) and \( y \) move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of maintaining a path between certain designated nodes. There are
two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes \( V \), and at a particular point in time there is a set \( E_0 \) of edges among these nodes. As the nodes move, the set of edges changes from \( E_0 \) to \( E_1 \), then to \( E_2 \), then to \( E_3 \), and so on, to an edge set \( E_b \). For \( i = 0, 1, 2, \ldots, b \), let \( G_i \) denote the graph \( (V, E_i) \). So if we were to watch the structure of the network on the nodes \( V \) as a “time lapse,” it would look precisely like the sequence of graphs \( G_0, G_1, G_2, \ldots, G_b \). We will assume that each of these graphs \( G_i \) is connected.

Now consider two particular nodes \( s, t \in V \). For an \( s-t \) path \( P \) in one of the graphs \( G_i \), we define the **length** of \( P \) to be simply the number of edges in \( P \), and we denote this \( \ell(P) \). Our goal is to produce a sequence of paths \( P_0, P_1, \ldots, P_b \) so that for each \( i \), \( P_i \) is an \( s-t \) path in \( G_i \). We want the paths to be relatively short. We also do not want there to be too many **changes**—points at which the identity of the path switches. Formally, we define changes\((P_0, P_1, \ldots, P_b)\) to be the number of indices \( i \) \((0 \leq i \leq b-1)\) for which \( P_i \neq P_{i+1} \).

Fix a constant \( K > 0 \). We define the **cost** of the sequence of paths \( P_0, P_1, \ldots, P_b \) to be

\[
\text{cost}(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot \text{changes}(P_0, P_1, \ldots, P_b).
\]

(a) Suppose it is possible to choose a single path \( P \) that is an \( s-t \) path in each of the graphs \( G_0, G_1, \ldots, G_b \). Give a polynomial-time algorithm to find the shortest such path.

(b) Give a polynomial-time algorithm to find a sequence of paths \( P_0, P_1, \ldots, P_b \) of minimum cost, where \( P_i \) is an \( s-t \) path in \( G_i \) for \( i = 0, 1, \ldots, b \).

15. On most clear days, a group of your friends in the Astronomy Department gets together to plan out the astronomical events they’re going to try observing that night. We’ll make the following assumptions about the events.

- There are \( n \) events, which for simplicity we’ll assume occur in sequence separated by exactly one minute each. Thus event \( j \) occurs at minute \( j \); if they don’t observe this event at exactly minute \( j \), then they miss out on it.

For any query, contact us at help@tutorialsduniya.com
Chapter 6  Dynamic Programming

- The sky is mapped according to a one-dimensional coordinate system (measured in degrees from some central baseline); event \( j \) will be taking place at coordinate \( d_j \), for some integer value \( d_j \). The telescope starts at coordinate 0 at minute 0.

- The last event, \( n \), is much more important than the others; so it is required that they observe event \( n \).

The Astronomy Department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one degree per minute. Thus they do not expect to be able to observe all \( n \) events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event \( n \) must be observed.

We say that a subset \( S \) of the events is **viewable** if it is possible to observe each event \( j \in S \) at its appointed time \( j \), and the telescope has adequate time (moving at its maximum of one degree per minute) to move between consecutive events in \( S \).

**The problem.** Given the coordinates of each of the \( n \) events, find a viewable subset of maximum size, subject to the requirement that it should contain event \( n \). Such a solution will be called **optimal**.

**Example.** Suppose the one-dimensional coordinates of the events are as shown here.

<table>
<thead>
<tr>
<th>Event</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinate</td>
<td>1</td>
<td>-4</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>-4</td>
<td>6</td>
<td>7</td>
<td>-2</td>
</tr>
</tbody>
</table>

Then the optimal solution is to observe events 1, 3, 6, 9. Note that the telescope has time to move from one event in this set to the next, even moving at one degree per minute.

**Example.** Suppose the one-dimensional coordinates of the events are as shown here.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

Mark all events \( j \) with \(|d_n - d_j| > n - j\) as illegal (as observing them would prevent you from observing event \( n \))

Mark all other events as legal

Initialize current position to coordinate 0 at minute 0

While not at end of event sequence

Find the earliest legal event \( j \) that can be reached without exceeding the maximum movement rate of the telescope

Add \( j \) to the set \( S \)
Exercises

Update current position to be coord. \(-d_j\) at minute \(j\)
Endwhile
Output the set \(S\)

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for the coordinates \(d_1, d_2, \ldots, d_n\) of the events and returns the size of an optimal solution.

16. There are many sunny days in Ithaca, New York; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on a rainy day. The ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism she uses to do this.

Each ROTC person on campus except the ranking officer reports to a unique superior officer. Thus the reporting hierarchy can be described by a tree \(T\), rooted at the ranking officer, in which each other node \(v\) has a parent node \(u\) equal to his or her superior officer. Conversely, we will call \(v\) a direct subordinate of \(u\). See Figure 6.30, in which A is the ranking officer, B and D are the direct subordinates of A, and C is the direct subordinate of B.

To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone; for example, in Figure 6.30, A would not be allowed to call C.

We can picture this process as being divided into rounds. In one round, each person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates. For example, in Figure 6.30, it will take only two rounds if A starts by calling B, but it will take three rounds if A starts by calling D.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

17. Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They’ve defined something called a rising trend, as follows.
Chapter 6  Dynamic Programming

They have the closing price for a given stock recorded for \( n \) days in succession; let these prices be denoted \( P[1], P[2], \ldots, P[n] \). A rising trend in these prices is a subsequence of the prices \( P[i_1], P[i_2], \ldots, P[i_k] \), for days \( i_1 < i_2 < \ldots < i_k \), so that

- \( i_1 = 1 \), and
- \( P[i_j] < P[i_{j+1}] \) for each \( j = 1, 2, \ldots, k - 1 \).

Thus a rising trend is a subsequence of the days—beginning on the first day and not necessarily contiguous—so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

**Example.** Suppose \( n = 7 \), and the sequence of prices is

\[
10, 1, 2, 11, 3, 4, 12.
\]

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

(a) Show that the following algorithm does not correctly return the length of the longest rising trend, by giving an instance on which it fails to return the correct answer.

Define \( i = 1 \)
\[
L = 1
\]
For \( j = 2 \) to \( n \)
If \( P[j] > P[i] \) then
Set \( i = j \).
Add 1 to \( L \).
Endif
Endfor

In your example, give the actual length of the longest rising trend, and say what the algorithm above returns.

(b) Give an efficient algorithm that takes a sequence of prices \( P[1], P[2], \ldots, P[n] \) and returns the length of the longest rising trend.

18. Consider the sequence alignment problem over a four-letter alphabet \( \{z_1, z_2, z_3, z_4\} \), with a given gap cost and given mismatch costs. Assume that each of these parameters is a positive integer.
Suppose you are given two strings $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$ and a proposed alignment between them. Give an $O(mn)$ algorithm to decide whether this alignment is the unique minimum-cost alignment between $A$ and $B$.

19. You’re consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: “untangling” a superposition of two known signals. Specifically, they’re picturing a situation in which each of two ships is emitting a short sequence of 0s and is over and over, and they want to make sure that the signal they’re hearing is simply an interleaving of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string $x$ consisting of 0s and 1s, we write $x^k$ to denote $k$ copies of $x$ concatenated together. We say that a string $x'$ is a repetition of $x$ if it is a prefix of $x^k$ for some number $k$. So $x' = 10101010101$ is a repetition of $x = 101$.

We say that a string $s$ is an interleaving of $x$ and $y$ if its symbols can be partitioned into two (not necessarily contiguous) subsequences $s'$ and $s''$, so that $s'$ is a repetition of $x$ and $s''$ is a repetition of $y$. (So each symbol in $s$ must belong to exactly one of $s'$ or $s''$.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of $x$ and $y$, since characters 1,2,5,7,8,9 form 10101—a repetition of $x$—and the remaining characters 3,4,6 form 000—a repetition of $y$.

In terms of our application, $x$ and $y$ are the repeating sequences from the two ships, and $s$ is the signal we’re listening to: We want to make sure $s$ “unravels” into simple repetitions of $x$ and $y$. Give an efficient algorithm that takes strings $s$, $x$, and $y$ and decides if $s$ is an interleaving of $x$ and $y$.

20. Suppose it’s nearing the end of the semester and you’re taking $n$ courses, each with a final project that still has to be done. Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the $n$ projects.

You have a total of $H > n$ hours in which to work on the $n$ projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume $H$ is a positive integer, and you’ll spend an integer number of hours on each project. To figure out how best to divide up your time, you’ve come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$ (rough
estimates, of course) for each of your \( n \) courses; if you spend \( h \leq H \) hours on the project for course \( i \), you'll get a grade of \( f_i(h) \). (You may assume that the functions \( f_i \) are nondecreasing: if \( h < h' \), then \( f_i(h) \leq f_i(h') \).)

So the problem is: Given these functions \( \{f_i\} \), decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the \( f_i \), is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in \( n \), \( g \), and \( H \); none of these quantities should appear as an exponent in your running time.

21. Some time back, you helped a group of friends who were doing simulations for a computation-intensive investment company, and they've come back to you with a new problem. They're looking at \( n \) consecutive days of a given stock, at some point in the past. The days are numbered \( i = 1, 2, \ldots, n \); for each day \( i \), they have a price \( p(i) \) per share for the stock on that day.

For certain (possibly large) values of \( k \), they want to study what they call \( k \)-shot strategies. A \( k \)-shot strategy is a collection of \( m \) pairs of days \( (b_1, s_1), \ldots, (b_m, s_m) \), where \( 0 \leq m \leq k \) and

\[
1 \leq b_1 < s_1 < b_2 < s_2 \cdots < b_m < s_m \leq n.
\]

We view these as a set of up to \( k \) nonoverlapping intervals, during each of which the investors buy 1,000 shares of the stock (on day \( b_i \)) and then sell it (on day \( s_i \)). The return of a given \( k \)-shot strategy is simply the profit obtained from the \( m \) buy-sell transactions, namely,

\[
1,000 \sum_{i=1}^{m} p(s_i) - p(b_i).
\]

The investors want to assess the value of \( k \)-shot strategies by running simulations on their \( n \)-day trace of the stock price. Your goal is to design an efficient algorithm that determines, given the sequence of prices, the \( k \)-shot strategy with the maximum possible return. Since \( k \) may be relatively large in these simulations, your running time should be polynomial in both \( n \) and \( k \); it should not contain \( k \) in the exponent.

22. To assess how “well-connected” two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the number of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph \( G = (V, E) \), with costs on the edges; the costs may be positive or
negative, but every cycle in the graph has strictly positive cost. We are also given two nodes \( v, w \in V \). Give an efficient algorithm that computes the number of shortest \( v-w \) paths in \( G \). (The algorithm should not list all the paths; just the number suffices.)

23. Suppose you are given a directed graph \( G = (V, E) \) with costs on the edges \( c_e \) for \( e \in E \) and a sink \( t \) (costs may be negative). Assume that you also have finite values \( d(v) \) for \( v \in V \). Someone claims that, for each node \( v \in V \), the quantity \( d(v) \) is the cost of the minimum-cost path from node \( v \) to the sink \( t \).

(a) Give a linear-time algorithm (time \( O(m) \) if the graph has \( m \) edges) that verifies whether this claim is correct.

(b) Assume that the distances are correct, and \( d(v) \) is finite for all \( v \in V \).
Now you need to compute distances to a different sink \( t' \). Give an \( O(m \log n) \) algorithm for computing distances \( d'(v) \) for all nodes \( v \in V \) to the sink node \( t' \). (Hint: It is useful to consider a new cost function defined as follows: for edge \( e = (v, w) \), let \( c'_e = c_e - d(v) + d(w) \). Is there a relation between costs of paths for the two different costs \( c \) and \( c' \)?)

24. Gerrymandering is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. Recent court challenges to the practice have argued that through this calculated redistricting, large numbers of voters are being effectively (and intentionally) disenfranchised.

Computers, it turns out, have been implicated as the source of some of the “villainy” in the news coverage on this topic: Thanks to powerful software, gerrymandering has changed from an activity carried out by a bunch of people with maps, pencil, and paper into the industrial-strength process that it is today. Why is gerrymandering a computational problem? There are database issues involved in tracking voter demographics down to the level of individual streets and houses; and there are algorithmic issues involved in grouping voters into districts. Let’s think a bit about what these latter issues look like.

Suppose we have a set of \( n \) precincts \( P_1, P_2, \ldots, P_n \), each containing \( m \) registered voters. We're supposed to divide these precincts into two districts, each consisting of \( n/2 \) of the precincts. Now, for each precinct, we have information on how many voters are registered to each of two political parties. (Suppose, for simplicity, that every voter is registered to one of these two.) We'll say that the set of precincts is susceptible to gerrymandering if it is possible to perform the division into two districts in such a way that the same party holds a majority in both districts.
Give an algorithm to determine whether a given set of precincts is susceptible to gerrymandering; the running time of your algorithm should be polynomial in \( n \) and \( m \).

**Example.** Suppose we have \( n = 4 \) precincts, and the following information on registered voters.

<table>
<thead>
<tr>
<th>Precinct</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Party A</td>
<td>55</td>
<td>43</td>
<td>60</td>
<td>47</td>
</tr>
<tr>
<td>Party B</td>
<td>45</td>
<td>57</td>
<td>40</td>
<td>53</td>
</tr>
</tbody>
</table>

This set of precincts is susceptible since, if we grouped precincts 1 and 4 into one district, and precincts 2 and 3 into the other, then party A would have a majority in both districts. (Presumably, the “we” who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: Although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

25. Consider the problem faced by a stockbroker trying to sell a large number of shares of stock in a company whose stock price has been steadily falling in value. It is always hard to predict the right moment to sell stock, but owning a lot of shares in a single company adds an extra complication: the mere act of selling many shares in a single day will have an adverse effect on the price.

Since future market prices, and the effect of large sales on these prices, are very hard to predict, brokerage firms use models of the market to help them make such decisions. In this problem, we will consider the following simple model. Suppose we need to sell \( x \) shares of stock in a company, and suppose that we have an accurate model of the market: it predicts that the stock price will take the values \( p_1, p_2, \ldots, p_n \) over the next \( n \) days. Moreover, there is a function \( f(\cdot) \) that predicts the effect of large sales: if we sell \( y \) shares on a single day, it will permanently decrease the price by \( f(y) \) from that day onward. So, if we sell \( y_1 \) shares on day 1, we obtain a price per share of \( p_1 - f(y_1) \), for a total income of \( y_1 \cdot (p_1 - f(y_1)) \). Having sold \( y_1 \) shares on day 1, we can then sell \( y_2 \) shares on day 2 for a price per share of \( p_2 - f(y_1) - f(y_2) \); this yields an additional income of \( y_2 \cdot (p_2 - f(y_1) - f(y_2)) \). This process continues over all \( n \) days. (Note, as in our calculation for day 2, that the decreases from earlier days are absorbed into the prices for all later days.)

Design an efficient algorithm that takes the prices \( p_1, \ldots, p_n \) and the function \( f(\cdot) \) (written as a list of values \( f(1), f(2), \ldots, f(x) \)) and determines
the best way to sell \( x \) shares by day \( n \). In other words, find natural numbers \( y_1, y_2, \ldots, y_n \) so that \( x = y_1 + \ldots + y_n \), and selling \( y_i \) shares on day \( i \) for \( i = 1, 2, \ldots, n \) maximizes the total income achievable. You should assume that the share value \( p_i \) is monotone decreasing, and \( f(\cdot) \) is monotone increasing; that is, selling a larger number of shares causes a larger drop in the price. Your algorithm's running time can have a polynomial dependence on \( n \) (the number of days), \( x \) (the number of shares), and \( p_1 \) (the peak price of the stock).

**Example** Consider the case when \( n = 3 \); the prices for the three days are 90, 80, 40; and \( f(y) = 1 \) for \( y \leq 40,000 \) and \( f(y) = 20 \) for \( y > 40,000 \). Assume you start with \( x = 100,000 \) shares. Selling all of them on day 1 would yield a price of 70 per share, for a total income of 7,000,000. On the other hand, selling 40,000 shares on day 1 yields a price of 89 per share, and selling the remaining 60,000 shares on day 2 results in a price of 59 per share, for a total income of 7,100,000.

26. Consider the following inventory problem. You are running a company that sells some large product (let's assume you sell trucks), and predictions tell you the quantity of sales to expect over the next \( n \) months. Let \( d_i \) denote the number of sales you expect in month \( i \). We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are stored until the beginning of the next month. You can store at most \( S \) trucks, and it costs \( C \) to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of \( K \) each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands \( \{d_i\} \), and minimize the costs. In summary:

- There are two parts to the cost: (1) storage—it costs \( C \) for every truck on hand that is not needed that month; (2) ordering fees—it costs \( K \) for every order placed.

- In each month you need enough trucks to satisfy the demand \( d_i \), but the number left over after satisfying the demand for the month should not exceed the inventory limit \( S \).

Give an algorithm that solves this problem in time that is polynomial in \( n \) and \( S \).

27. The owners of an independently operated gas station are faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to \( L \) gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order,
they need to pay a fixed price $P$ for delivery in addition to the cost of the gas ordered. However, it costs $c$ to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost.

They are planning to close for a week in the winter, and they want their tank to be empty by the time they close. Luckily, based on years of experience, they have accurate projections for how much gas they will need each day until this point in time. Assume that there are $n$ days left until they close, and they need $g_i$ gallons of gas for each of the days $i = 1, \ldots, n$. Assume that the tank is empty at the end of day 0. Give an algorithm to decide on which days they should place orders, and how much to order so as to minimize their total cost.

28. Recall the scheduling problem from Section 4.2 in which we sought to minimize the maximum lateness. There are $n$ jobs, each with a deadline $d_i$ and a required processing time $t_i$, and all jobs are available to be scheduled starting at time $s$. For a job $i$ to be done, it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and different jobs should be assigned nonoverlapping intervals. As usual, such an assignment of times will be called a schedule.

In this problem, we consider the same setup, but want to optimize a different objective. In particular, we consider the case in which each job must either be done by its deadline or not at all. We’ll say that a subset $J$ of the jobs is schedulable if there is a schedule for the jobs in $J$ so that each of them finishes by its deadline. Your problem is to select a schedulable subset of maximum possible size and give a schedule for this subset that allows each job to finish by its deadline.

(a) Prove that there is an optimal solution $J$ (i.e., a schedulable set of maximum size) in which the jobs in $J$ are scheduled in increasing order of their deadlines.

(b) Assume that all deadlines $d_i$ and required times $t_i$ are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of jobs $n$, and the maximum deadline $D = \max_i d_i$.

29. Let $G = (V, E)$ be a graph with $n$ nodes in which each pair of nodes is joined by an edge. There is a positive weight $w_{ij}$ on each edge $(i, j)$; and we will assume these weights satisfy the triangle inequality $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subseteq V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in $V'$.

We are given a set $X \subseteq V$ of $k$ terminals that must be connected by edges. We say that a Steiner tree on $X$ is a set $Z$ so that $X \subseteq Z \subseteq V$, together
with a spanning subtree $T$ of $G[Z]$. The weight of the Steiner tree is the weight of the tree $T$.

Show that there is function $f(\cdot)$ and a polynomial function $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on $X$ can be solved in time $O(f(k) \cdot p(n))$. 
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
</table>

TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from [https://www.tutorialsduniya.com](https://www.tutorialsduniya.com).

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
Chapter 12 showed that a binary search tree of height \( h \) can support any of the basic dynamic-set operations—such as \texttt{SEARCH}, \texttt{PREDECESSOR}, \texttt{SUCCESSOR}, \texttt{MINIMUM}, \texttt{MAXIMUM}, \texttt{INSERT}, and \texttt{DELETE}—in \( O(h) \) time. Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take \( O(\lg n) \) time in the worst case.

13.1 Properties of red-black trees

A \textit{red-black tree} is a binary search tree with one extra bit of storage per node: its \textit{color}, which can be either \texttt{RED} or \texttt{BLACK}. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately \textit{balanced}.

Each node of the tree now contains the attributes \texttt{color}, \texttt{key}, \texttt{left}, \texttt{right}, and \( p \). If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value \texttt{NIL}. We shall regard these \texttt{NIL}s as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following \textit{red-black properties}:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (\texttt{NIL}) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
Figure 13.1(a) shows an example of a red-black tree.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL (see page 238). For a red-black tree $T$, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its color attribute is BLACK, and its other attributes—$p$, $left$, $right$, and $key$—can take on arbitrary values. As Figure 13.1(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

We use the sentinel so that we can treat a NIL child of a node $x$ as an ordinary node whose parent is $x$. Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined, that approach would waste space. Instead, we use the one sentinel $T.nil$ to represent all the NILS—all leaves and the root’s parent. The values of the attributes $p$, $left$, $right$, and $key$ of the sentinel are immaterial, although we may set them during the course of a procedure for our convenience.

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. In the remainder of this chapter, we omit the leaves when we draw red-black trees, as shown in Figure 13.1(c).

We call the number of black nodes on any simple path from, but not including, a node $x$ down to a leaf the **black-height** of the node, denoted $bh(x)$. By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root.

The following lemma shows why red-black trees make good search trees.

**Lemma 13.1**
A red-black tree with $n$ internal nodes has height at most $2 \log(n + 1)$.

**Proof** We start by showing that the subtree rooted at any node $x$ contains at least $2^{bh(x)} - 1$ internal nodes. We prove this claim by induction on the height of $x$. If the height of $x$ is 0, then $x$ must be a leaf ($T.nil$), and the subtree rooted at $x$ indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node $x$ that has positive height and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of $x$ is less than the height of $x$ itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at $x$ contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let $h$ be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not
Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel $T.nil$, which is always black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
including the root, must be black. Consequently, the black-height of the root must be at least \( h/2 \); thus,

\[
n \geq 2^{h/2} - 1.
\]

Moving the 1 to the left-hand side and taking logarithms on both sides yields

\[
\log(n + 1) \geq h/2, \text{ or } h \leq 2\log(n + 1).
\]

As an immediate consequence of this lemma, we can implement the dynamic-set operations \textsc{Search}, \textsc{Minimum}, \textsc{Maximum}, \textsc{Successor}, and \textsc{Predecessor} in \( O(\log n) \) time on red-black trees, since each can run in \( O(h) \) time on a binary search tree of height \( h \) (as shown in Chapter 12) and any red-black tree on \( n \) nodes is a binary search tree with height \( O(\log n) \). (Of course, references to \textsc{NIL} in the algorithms of Chapter 12 would have to be replaced by \textsc{T.nil}.) Although the algorithms \textsc{Tree-Insert} and \textsc{Tree-Delete} from Chapter 12 run in \( O(\log n) \) time when given a red-black tree as input, they do not directly support the dynamic-set operations \textsc{Insert} and \textsc{Delete}, since they do not guarantee that the modified binary search tree will be a red-black tree. We shall see in Sections 13.3 and 13.4, however, how to support these two operations in \( O(\log n) \) time.

\section*{Exercises}

\subsection*{13.1-1}
In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys \( \{1, 2, \ldots, 15\} \). Add the \textsc{NIL} leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

\subsection*{13.1-2}
Draw the red-black tree that results after \textsc{Tree-Insert} is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

\subsection*{13.1-3}
Let us define a \textit{relaxed red-black tree} as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree \( T \) whose root is red. If we color the root of \( T \) black but make no other changes to \( T \), is the resulting tree a red-black tree?

\subsection*{13.1-4}
Suppose that we “absorb” every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all
its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

13.1-5
Show that the longest simple path from a node \( x \) in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node \( x \) to a descendant leaf.

13.1-6
What is the largest possible number of internal nodes in a red-black tree with black-height \( k \)? What is the smallest possible number?

13.1-7
Describe a red-black tree on \( n \) keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with \( n \) keys, take \( O(\lg n) \) time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through \textit{rotation}, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. When we do a left rotation on a node \( x \), we assume that its right child \( y \) is not \( T.nil \); \( x \) may be any node in the tree whose right child is not \( T.nil \). The left rotation “pivots” around the link from \( x \) to \( y \). It makes \( y \) the new root of the subtree, with \( x \) as \( y \)'s left child and \( y \)'s left child as \( x \)'s right child.

The pseudocode for LEFT-ROTATE assumes that \( x.right \neq T.nil \) and that the root’s parent is \( T.nil \).
13.2 Rotations

**Figure 13.2** The rotation operations on a binary search tree. The operation `LEFT-ROTATE(T, x)` transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation `RIGHT-ROTATE(T, y)` transforms the configuration on the left into the configuration on the right. The letters $\alpha$, $\beta$, and $\gamma$ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in $\alpha$ precede $x.key$, which precedes the keys in $\beta$, which precede $y.key$, which precedes the keys in $\gamma$.

**LEFT-ROTATE(T, x)**

1. $y = x.right$  // set $y$
2. $x.right = y.left$  // turn $y$’s left subtree into $x$’s right subtree
3. **if** $y.left \neq T.nil$
4. \hspace{1em} $y.left.p = x$
5. \hspace{1em} $y.p = x.p$  // link $x$’s parent to $y$
6. **if** $x.p == T.nil$
7. \hspace{1em} $T.root = y$
8. **else if** $x == x.p.left$
9. \hspace{1em} $x.p.left = y$
10. **else** $x.p.right = y$
11. \hspace{1em} $y.left = x$  // put $x$ on $y$’s left
12. $x.p = y$

Figure 13.3 shows an example of how `LEFT-ROTATE` modifies a binary search tree. The code for `RIGHT-ROTATE` is symmetric. Both `LEFT-ROTATE` and `RIGHT-ROTATE` run in $O(1)$ time. Only pointers are changed by a rotation; all other attributes in a node remain the same.

**Exercises**

13.2-1
Write pseudocode for `RIGHT-ROTATE`.

13.2-2
Argue that in every $n$-node binary search tree, there are exactly $n - 1$ possible rotations.
Figure 13.3 An example of how the procedure LEFT-ROTATE($T, x$) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

13.2-3
Let $a$, $b$, and $c$ be arbitrary nodes in subtrees $\alpha$, $\beta$, and $\gamma$, respectively, in the left tree of Figure 13.2. How do the depths of $a$, $b$, and $c$ change when a left rotation is performed on node $x$ in the figure?

13.2-4
Show that any arbitrary $n$-node binary search tree can be transformed into any other arbitrary $n$-node binary search tree using $O(n)$ rotations. (Hint: First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

13.2-5
We say that a binary search tree $T_1$ can be right-converted to binary search tree $T_2$ if it is possible to obtain $T_2$ from $T_1$ via a series of calls to RIGHT-ROTATE. Give an example of two trees $T_1$ and $T_2$ such that $T_1$ cannot be right-converted to $T_2$. Then, show that if a tree $T_1$ can be right-converted to $T_2$, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.
13.3 Insertion

We can insert a node into an \( n \)-node red-black tree in \( O(\lg n) \) time. To do so, we use a slightly modified version of the Tree-Insert procedure (Section 12.3) to insert node \( z \) into the tree \( T \) as if it were an ordinary binary search tree, and then we color \( z \) red. (Exercise 13.3-1 asks you to explain why we choose to make node \( z \) red rather than black.) To guarantee that the red-black properties are preserved, we then call an auxiliary procedure RB-Insert-Fixup to recolor nodes and perform rotations. The call RB-Insert(\( T, z \)) inserts node \( z \), whose key is assumed to have already been filled in, into the red-black tree \( T \).

\begin{verbatim}
RB-Insert(\( T, z \))
1  y = T.nil
2  x = T.root
3  while x \( \neq \) T.nil
4      y = x
5      if z.key < x.key
6          x = x.left
7      else x = x.right
8  z.p = y
9  if y == T.nil
10     T.root = z
11  elseif z.key < y.key
12     y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-Insert-Fixup(\( T, z \))
\end{verbatim}

The procedures Tree-Insert and RB-Insert differ in four ways. First, all instances of NIL in Tree-Insert are replaced by \( T.nil \). Second, we set \( z.left \) and \( z.right \) to \( T.nil \) in lines 14–15 of RB-Insert, in order to maintain the proper tree structure. Third, we color \( z \) red in line 16. Fourth, because coloring \( z \) red may cause a violation of one of the red-black properties, we call RB-Insert-Fixup(\( T, z \)) in line 17 of RB-Insert to restore the red-black properties.
RB-INSERT-FIXUP($T, z$)

1. while $z.p.color == RED$
2. if $z.p == z.p.p.left$
3. $y = z.p.p.right$
4. if $y.color == RED$
5. $z.p.color = BLACK$  // case 1
6. $y.color = BLACK$  // case 1
7. $z.p.p.color = RED$  // case 1
8. $z = z.p.p$  // case 1
9. else if $z == z.p.right$
10. $z = z.p$  // case 2
11. LEFT-ROTATE($T, z$)  // case 2
12. $z.p.color = BLACK$  // case 3
13. $z.p.p.color = RED$  // case 3
14. RIGHT-ROTATE($T, z.p.p$)  // case 3
15. else (same as then clause with “right” and “left” exchanged)
16. $T.root.color = BLACK$

To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps. First, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node $z$ is inserted and colored red. Second, we shall examine the overall goal of the while loop in lines 1–15. Finally, we shall explore each of the three cases\(^1\) within the while loop’s body and see how they accomplish the goal. Figure 13.4 shows how RB-INSERT-FIXUP operates on a sample red-black tree.

Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP? Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel $T.nil$. Property 5, which says that the number of black nodes is the same on every simple path from a given node, is satisfied as well, because node $z$ replaces the (black) sentinel, and node $z$ is red with sentinel children. Thus, the only properties that might be violated are property 2, which requires the root to be black, and property 4, which says that a red node cannot have a red child. Both possible violations are due to $z$ being colored red. Property 2 is violated if $z$ is the root, and property 4 is violated if $z$’s parent is red. Figure 13.4(a) shows a violation of property 4 after the node $z$ has been inserted.

\(^1\)Case 2 falls through into case 3, and so these two cases are not mutually exclusive.
13.3 Insertion

Figure 13.4 The operation of RB-INSERT-FIXUP. (a) A node \( z \) after insertion. Because both \( z \) and its parent \( z.p \) are red, a violation of property 4 occurs. Since \( z \)'s uncle \( y \) is red, case 1 in the code applies. We recolor nodes and move the pointer \( z \) up the tree, resulting in the tree shown in (b). Once again, \( z \) and its parent are both red, but \( z \)'s uncle \( y \) is black. Since \( z \) is the right child of \( z.p \), case 2 applies. We perform a left rotation, and the tree that results is shown in (c). Now, \( z \) is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.
The **while** loop in lines 1–15 maintains the following three-part invariant at the start of each iteration of the loop:

a. Node \( z \) is red.

b. If \( z.p \) is the root, then \( z.p \) is black.

c. If the tree violates any of the red-black properties, then it violates at most one of them, and the violation is of either property 2 or property 4. If the tree violates property 2, it is because \( z \) is the root and is red. If the tree violates property 4, it is because both \( z \) and \( z.p \) are red.

Part (c), which deals with violations of red-black properties, is more central to showing that \( \text{RB-INSERT-FIXUP} \) restores the red-black properties than parts (a) and (b), which we use along the way to understand situations in the code. Because we’ll be focusing on node \( z \) and nodes near it in the tree, it helps to know from part (a) that \( z \) is red. We shall use part (b) to show that the node \( z.p.p \) exists when we reference it in lines 2, 3, 7, 8, 13, and 14.

Recall that we need to show that a loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

We start with the initialization and termination arguments. Then, as we examine how the body of the loop works in more detail, we shall argue that the loop maintains the invariant upon each iteration. Along the way, we shall also demonstrate that each iteration of the loop has two possible outcomes: either the pointer \( z \) moves up the tree, or we perform some rotations and then the loop terminates.

**Initialization:** Prior to the first iteration of the loop, we started with a red-black tree with no violations, and we added a red node \( z \). We show that each part of the invariant holds at the time \( \text{RB-INSERT-FIXUP} \) is called:

a. When \( \text{RB-INSERT-FIXUP} \) is called, \( z \) is the red node that was added.

b. If \( z.p \) is the root, then \( z.p \) started out black and did not change prior to the call of \( \text{RB-INSERT-FIXUP} \).

c. We have already seen that properties 1, 3, and 5 hold when \( \text{RB-INSERT-FIXUP} \) is called.

If the tree violates property 2, then the red root must be the newly added node \( z \), which is the only internal node in the tree. Because the parent and both children of \( z \) are the sentinel, which is black, the tree does not also violate property 4. Thus, this violation of property 2 is the only violation of red-black properties in the entire tree.

If the tree violates property 4, then, because the children of node \( z \) are black sentinels and the tree had no other violations prior to \( z \) being added, the
violation must be because both \( z \) and \( z.p \) are red. Moreover, the tree violates no other red-black properties.

**Termination:** When the loop terminates, it does so because \( z.p \) is black. (If \( z \) is the root, then \( z.p \) is the sentinel \( T.nil \), which is black.) Thus, the tree does not violate property 4 at loop termination. By the loop invariant, the only property that might fail to hold is property 2. Line 16 restores this property, too, so that when RB-INSERT-FIXUP terminates, all the red-black properties hold.

**Maintenance:** We actually need to consider six cases in the while loop, but three of them are symmetric to the other three, depending on whether line 2 determines \( z \)'s parent \( z.p \) to be a left child or a right child of \( z \)'s grandparent \( z.p.p \). We have given the code only for the situation in which \( z.p \) is a left child. The node \( z.p.p \) exists, since by part (b) of the loop invariant, if \( z.p \) is the root, then \( z.p \) is black. Since we enter a loop iteration only if \( z.p \) is red, we know that \( z.p \) cannot be the root. Hence, \( z.p.p \) exists.

We distinguish case 1 from cases 2 and 3 by the color of \( z \)'s parent's sibling, or “uncle.” Line 3 makes \( y \) point to \( z \)'s uncle \( z.p.p.right \), and line 4 tests \( y \)'s color. If \( y \) is red, then we execute case 1. Otherwise, control passes to cases 2 and 3. In all three cases, \( z \)'s grandparent \( z.p.p \) is black, since its parent \( z.p \) is red, and property 4 is violated only between \( z \) and \( z.p \).

**Case 1: z’s uncle y is red**

Figure 13.5 shows the situation for case 1 (lines 5–8), which occurs when both \( z.p \) and \( y \) are red. Because \( z.p.p \) is black, we can color both \( z.p \) and \( y \) black, thereby fixing the problem of \( z \) and \( z.p \) both being red, and we can color \( z.p.p \) red, thereby maintaining property 5. We then repeat the while loop with \( z.p.p \) as the new node \( z \). The pointer \( z \) moves up two levels in the tree.

Now, we show that case 1 maintains the loop invariant at the start of the next iteration. We use \( z \) to denote node \( z \) in the current iteration, and \( z' = z.p.p \) to denote the node that will be called node \( z \) at the test in line 1 upon the next iteration.

a. Because this iteration colors \( z.p.p \) red, node \( z' \) is red at the start of the next iteration.

b. The node \( z'.p \) is \( z.p.p.p \) in this iteration, and the color of this node does not change. If this node is the root, it was black prior to this iteration, and it remains black at the start of the next iteration.

c. We have already argued that case 1 maintains property 5, and it does not introduce a violation of properties 1 or 3.
Case 1 of the procedure RB-INSERT-FIXUP. Property 4 is violated, since \( z \) and its parent \( z.p \) are both red. We take the same action whether (a) \( z \) is a right child or (b) \( z \) is a left child. Each of the subtrees \( \alpha, \beta, \gamma, \delta, \) and \( \epsilon \) has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks. The \textbf{while} loop continues with node \( z \)'s grandparent \( z.p.p \) as the new \( z \). Any violation of property 4 can now occur only between the new \( z \), which is red, and its parent, if it is red as well.

If node \( z' \) is the root at the start of the next iteration, then case 1 corrected the lone violation of property 4 in this iteration. Since \( z' \) is red and it is the root, property 2 becomes the only one that is violated, and this violation is due to \( z' \).

If node \( z' \) is not the root at the start of the next iteration, then case 1 has not created a violation of property 2. Case 1 corrected the lone violation of property 4 that existed at the start of this iteration. It then made \( z' \) red and left \( z'.p \) alone. If \( z'.p \) was black, there is no violation of property 4. If \( z'.p \) was red, coloring \( z' \) red created one violation of property 4 between \( z' \) and \( z'.p \).

**Case 2: \( z \)'s uncle \( y \) is black and \( z \) is a right child**

**Case 3: \( z \)'s uncle \( y \) is black and \( z \) is a left child**

In cases 2 and 3, the color of \( z \)'s uncle \( y \) is black. We distinguish the two cases according to whether \( z \) is a right or left child of \( z.p \). Lines 10–11 constitute case 2, which is shown in Figure 13.6 together with case 3. In case 2, node \( z \) is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 (lines 12–14), in which node \( z \) is a left child. Because
both \( z \) and \( z.p \) are red, the rotation affects neither the black-height of nodes nor property 5. Whether we enter case 3 directly or through case 2, \( z \)'s uncle \( y \) is black, since otherwise we would have executed case 1. Additionally, the node \( z.p.p \) exists, since we have argued that this node existed at the time that lines 2 and 3 were executed, and after moving \( z \) up one level in line 10 and then down one level in line 11, the identity of \( z.p.p \) remains unchanged. In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done. The while loop does not iterate another time, since \( z.p \) is now black.

We now show that cases 2 and 3 maintain the loop invariant. (As we have just argued, \( z.p \) will be black upon the next test in line 1, and the loop body will not execute again.)

a. Case 2 makes \( z \) point to \( z.p \), which is red. No further change to \( z \) or its color occurs in cases 2 and 3.

b. Case 3 makes \( z.p \) black, so that if \( z.p \) is the root at the start of the next iteration, it is black.

c. As in case 1, properties 1, 3, and 5 are maintained in cases 2 and 3. Since node \( z \) is not the root in cases 2 and 3, we know that there is no violation of property 2. Cases 2 and 3 do not introduce a violation of property 2, since the only node that is made red becomes a child of a black node by the rotation in case 3. Cases 2 and 3 correct the lone violation of property 4, and they do not introduce another violation.
Having shown that each iteration of the loop maintains the invariant, we have shown that RB-INSERT-FIXUP correctly restores the red-black properties.

**Analysis**

What is the running time of RB-INSERT? Since the height of a red-black tree on \( n \) nodes is \( O(\lg n) \), lines 1–16 of RB-INSERT take \( O(\lg n) \) time. In RB-INSERT-FIXUP, the while loop repeats only if case 1 occurs, and then the pointer \( z \) moves two levels up the tree. The total number of times the while loop can be executed is therefore \( O(\lg n) \). Thus, RB-INSERT takes a total of \( O(\lg n) \) time. Moreover, it never performs more than two rotations, since the while loop terminates if case 2 or case 3 is executed.

**Exercises**

13.3-1
In line 16 of RB-INSERT, we set the color of the newly inserted node \( z \) to red. Observe that if we had chosen to set \( z \)'s color to black, then property 4 of a red-black tree would not be violated. Why didn’t we choose to set \( z \)'s color to black?

13.3-2
Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

13.3-3
Suppose that the black-height of each of the subtrees \( \alpha, \beta, \gamma, \delta, \epsilon \) in Figures 13.5 and 13.6 is \( k \). Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

13.3-4
Professor Teach is concerned that RB-INSERT-FIXUP might set \( T.nil.color \) to \( \text{RED} \), in which case the test in line 1 would not cause the loop to terminate when \( z \) is the root. Show that the professor’s concern is unfounded by arguing that RB-INSERT-FIXUP never sets \( T.nil.color \) to \( \text{RED} \).

13.3-5
Consider a red-black tree formed by inserting \( n \) nodes with RB-INSERT. Argue that if \( n > 1 \), the tree has at least one red node.

13.3-6
Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.
13.4 Deletion

Like the other basic operations on an \( n \)-node red-black tree, deletion of a node takes time \( O(\lg n) \). Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the TREE-DELETE procedure (Section 12.3). First, we need to customize the TRANSPLANT subroutine that TREE-DELETE calls so that it applies to a red-black tree:

\[
\text{RB-TRANSPLANT}(T, u, v) \\
1 \quad \textbf{if} \ u.p == T.nil \\
2 \quad T.root = v \\
3 \quad \textbf{elseif} \ u == u.p.left \\
4 \quad u.p.left = v \\
5 \quad \textbf{else} \ u.p.right = v \\
6 \quad v.p = u.p
\]

The procedure RB-TRANSPLANT differs from TRANSPLANT in two ways. First, line 1 references the sentinel \( T.nil \) instead of \( \text{NIL} \). Second, the assignment to \( v.p \) in line 6 occurs unconditionally: we can assign to \( v.p \) even if \( v \) points to the sentinel. In fact, we shall exploit the ability to assign to \( v.p \) when \( v = T.nil \).

The procedure RB-DELETE is like the TREE-DELETE procedure, but with additional lines of pseudocode. Some of the additional lines keep track of a node \( y \) that might cause violations of the red-black properties. When we want to delete node \( z \) and \( z \) has fewer than two children, then \( z \) is removed from the tree, and we want \( y \) to be \( z \). When \( z \) has two children, then \( y \) should be \( z \)'s successor, and \( y \) moves into \( z \)'s position in the tree. We also remember \( y \)'s color before it is removed from or moved within the tree, and we keep track of the node \( x \) that moves into \( y \)'s original position in the tree, because node \( x \) might also cause violations of the red-black properties. After deleting node \( z \), RB-DELETE calls an auxiliary procedure RB-DELETE-FIXUP, which changes colors and performs rotations to restore the red-black properties.
RB-DELETE \((T, z)\)

1. \(y = z\)
2. \(y\text{-original-color} = y\text{.color}\)
3. \(\text{if} \ z\text{.left} == T\text{.nil}\)
   4. \(x = z\text{.right}\)
   5. RB-TRANSPLANT \((T, z, z\text{.right})\)
4. \(\text{elseif} \ z\text{.right} == T\text{.nil}\)
   5. \(x = z\text{.left}\)
   6. RB-TRANSPLANT \((T, z, z\text{.left})\)
7. \(\text{else} \ y = \text{TREE-MINIMUM}(z\text{.right})\)
8. \(y\text{-original-color} = y\text{.color}\)
9. \(x = y\text{.right}\)
10. \(\text{if} \ y\text{.p} == z\)
   11. \(x\text{.p} = y\)
   12. \(\text{else} \ \text{RB-TRANSPLANT}(T, y, y\text{.right})\)
   13. \(y\text{.right} = z\text{.right}\)
   14. \(y\text{.right.p} = y\)
   15. RB-TRANSPLANT \((T, z, y)\)
   16. \(y\text{.left} = z\text{.left}\)
   17. \(y\text{.left.p} = y\)
   18. \(y\text{.color} = z\text{.color}\)
19. \(\text{if} \ y\text{-original-color} == \text{BLACK}\)
   20. \(\text{RB-DELETE-FIXUP}(T, x)\)

Although RB-DELETE contains almost twice as many lines of pseudocode as TREE-DELETE, the two procedures have the same basic structure. You can find each line of TREE-DELETE within RB-DELETE (with the changes of replacing NIL by \(T\text{.nil}\) and replacing calls to TRANSPLANT by calls to RB-TRANSPLANT), executed under the same conditions.

Here are the other differences between the two procedures:

- We maintain node \(y\) as the node either removed from the tree or moved within the tree. Line 1 sets \(y\) to point to node \(z\) when \(z\) has fewer than two children and is therefore removed. When \(z\) has two children, line 9 sets \(y\) to point to \(z\)’s successor, just as in TREE-DELETE, and \(y\) will move into \(z\)’s position in the tree.

- Because node \(y\)’s color might change, the variable \(y\text{-original-color}\) stores \(y\)’s color before any changes occur. Lines 2 and 10 set this variable immediately after assignments to \(y\). When \(z\) has two children, then \(y \neq z\) and node \(y\) moves into node \(z\)’s original position in the red-black tree; line 20 gives \(y\) the same color as \(z\). We need to save \(y\)’s original color in order to test it at the
13.4 Deletion

end of RB-DELETE; if it was black, then removing or moving $y$ could cause violations of the red-black properties.

- As discussed, we keep track of the node $x$ that moves into node $y$’s original position. The assignments in lines 4, 7, and 11 set $x$ to point to either $y$’s only child or, if $y$ has no children, the sentinel $T.nil$. (Recall from Section 12.3 that $y$ has no left child.)

- Since node $x$ moves into node $y$’s original position, the attribute $x.p$ is always set to point to the original position in the tree of $y$’s parent, even if $x$ is, in fact, the sentinel $T.nil$. Unless $z$ is $y$’s original parent (which occurs only when $z$ has two children and its successor $y$ is $z$’s right child), the assignment to $x.p$ takes place in line 6 of RB-TRANSPLANT. (Observe that when RB-TRANSPLANT is called in lines 5, 8, or 14, the second parameter passed is the same as $x$.)

When $y$’s original parent is $z$, however, we do not want $x.p$ to point to $y$’s original parent, since we are removing that node from the tree. Because node $y$ will move up to take $z$’s position in the tree, setting $x.p$ to $y$ in line 13 causes $x.p$ to point to the original position of $y$’s parent, even if $x = T.nil$.

- Finally, if node $y$ was black, we might have introduced one or more violations of the red-black properties, and so we call RB-DELETE-FIXUP in line 22 to restore the red-black properties. If $y$ was red, the red-black properties still hold when $y$ is removed or moved, for the following reasons:

  1. No black-heights in the tree have changed.
  2. No red nodes have been made adjacent. Because $y$ takes $z$’s place in the tree, along with $z$’s color, we cannot have two adjacent red nodes at $y$’s new position in the tree. In addition, if $y$ was not $z$’s right child, then $y$’s original right child $x$ replaces $y$ in the tree. If $y$ is red, then $x$ must be black, and so replacing $y$ by $x$ cannot cause two red nodes to become adjacent.
  3. Since $y$ could not have been the root if it was red, the root remains black.

If node $y$ was black, three problems may arise, which the call of RB-DELETE-FIXUP will remedy. First, if $y$ had been the root and a red child of $y$ becomes the new root, we have violated property 2. Second, if both $x$ and $x.p$ are red, then we have violated property 4. Third, moving $y$ within the tree causes any simple path that previously contained $y$ to have one fewer black node. Thus, property 5 is now violated by any ancestor of $y$ in the tree. We can correct the violation of property 5 by saying that node $x$, now occupying $y$’s original position, has an “extra” black. That is, if we add 1 to the count of black nodes on any simple path that contains $x$, then under this interpretation, property 5 holds. When we remove or move the black node $y$, we “push” its blackness onto node $x$. The problem is that now node $x$ is neither red nor black, thereby violating property 1. Instead,
node $x$ is either “doubly black” or “red-and-black,” and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing $x$. The color attribute of $x$ will still be either RED (if $x$ is red-and-black) or BLACK (if $x$ is doubly black). In other words, the extra black on a node is reflected in $x$’s pointing to the node rather than in the color attribute.

We can now see the procedure RB-DELETE-FIXUP and examine how it restores the red-black properties to the search tree.

**RB-DELETE-FIXUP** $(T, x)$

```plaintext
1. while $x \neq T.root$ and $x.color == \text{BLACK}$
2.    if $x == x.p.left$
3.        $w = x.p.right$
4.        if $w.color == \text{RED}$
5.            $w.color = \text{BLACK}$  // case 1
6.            $x.p.color = \text{RED}$  // case 1
7.            LEFT-ROTATE$(T, x.p)$  // case 1
8.            $w = x.p.right$  // case 1
9.        if $w.left.color == \text{BLACK}$ and $w.right.color == \text{BLACK}$
10.           $w.color = \text{RED}$  // case 2
11.           $x = x.p$  // case 2
12.    else if $w.right.color == \text{BLACK}$
13.        $w.left.color = \text{BLACK}$  // case 3
14.        $w.color = \text{RED}$  // case 3
15.        RIGHT-ROTATE$(T, w)$  // case 3
16.        $w = x.p.right$  // case 3
17.        $w.color = x.p.color$  // case 4
18.        $x.p.color = \text{BLACK}$  // case 4
19.        $w.right.color = \text{BLACK}$  // case 4
20.        LEFT-ROTATE$(T, x.p)$  // case 4
21.    else (same as then clause with “right” and “left” exchanged)
22.        $x.color = \text{BLACK}$  // case 4
23.```

The procedure RB-DELETE-FIXUP restores properties 1, 2, and 4. Exercises 13.4-1 and 13.4-2 ask you to show that the procedure restores properties 2 and 4, and so in the remainder of this section, we shall focus on property 1. The goal of the while loop in lines 1–22 is to move the extra black up the tree until

1. $x$ points to a red-and-black node, in which case we color $x$ (singly) black in line 23;
2. $x$ points to the root, in which case we simply “remove” the extra black; or
3. having performed suitable rotations and recolorings, we exit the loop.
Within the **while** loop, \( x \) always points to a nonroot doubly black node. We determine in line 2 whether \( x \) is a left child or a right child of its parent \( x.p \). (We have given the code for the situation in which \( x \) is a left child; the situation in which \( x \) is a right child—line 22—is symmetric.) We maintain a pointer \( w \) to the sibling of \( x \). Since node \( x \) is doubly black, node \( w \) cannot be \( T.nil \), because otherwise, the number of blacks on the simple path from \( x.p \) to the (singly black) leaf \( w \) would be smaller than the number on the simple path from \( x.p \) to \( x \).

The four cases\(^2\) in the code appear in Figure 13.7. Before examining each case in detail, let’s look more generally at how we can verify that the transformation in each of the cases preserves property 5. The key idea is that in each case, the transformation applied preserves the number of black nodes (including \( x \)’s extra black) from (and including) the root of the subtree shown to each of the subtrees \( \alpha, \beta, \ldots, \zeta \). Thus, if property 5 holds prior to the transformation, it continues to hold afterward. For example, in Figure 13.7(a), which illustrates case 1, the number of black nodes from the root to either subtree \( \alpha \) or \( \beta \) is 3, both before and after the transformation. (Again, remember that node \( x \) adds an extra black.) Similarly, the number of black nodes from the root to any of \( \gamma, \delta, \varepsilon, \) and \( \zeta \) is 2, both before and after the transformation. In Figure 13.7(b), the counting must involve the value \( c \) of the \textit{color} attribute of the root of the subtree shown, which can be either \texttt{RED} or \texttt{BLACK}. If we define \( \text{count(RED)} = 0 \) and \( \text{count(BLACK)} = 1 \), then the number of black nodes from the root to \( \alpha \) is \( 2 + \text{count}(c) \), both before and after the transformation. In this case, after the transformation, the new node \( x \) has \textit{color} attribute \( c \), but this node is really either red-and-black (if \( c = \texttt{RED} \)) or doubly black (if \( c = \texttt{BLACK} \)). You can verify the other cases similarly (see Exercise 13.4-5).

**Case 1: \( x \)’s sibling \( w \) is red**

Case 1 (lines 5–8 of RB-DELETE-FIXUP and Figure 13.7(a)) occurs when node \( w \), the sibling of node \( x \), is red. Since \( w \) must have black children, we can switch the colors of \( w \) and \( x.p \) and then perform a left-rotation on \( x.p \) without violating any of the red-black properties. The new sibling of \( x \), which is one of \( w \)’s children prior to the rotation, is now black, and thus we have converted case 1 into case 2, 3, or 4.

Cases 2, 3, and 4 occur when node \( w \) is black; they are distinguished by the colors of \( w \)’s children.

---

\(^2\)As in RB-INSERT-FIXUP, the cases in RB-DELETE-FIXUP are not mutually exclusive.
Case 2: x’s sibling w is black, and both of w’s children are black
In case 2 (lines 10–11 of RB-DELETE-FIXUP and Figure 13.7(b)), both of w’s children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to x.p, which was originally either red or black. We do so by repeating the while loop with x.p as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original x.p was red. Hence, the value c of the color attribute of the new node x is RED, and the loop terminates when it tests the loop condition. We then color the new node x (singly) black in line 23.

Case 3: x’s sibling w is black, w’s left child is red, and w’s right child is black
Case 3 (lines 13–16 and Figure 13.7(c)) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child w.left and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.

Case 4: x’s sibling w is black, and w’s right child is red
Case 4 (lines 17–21 and Figure 13.7(d)) occurs when node x’s sibling w is black and w’s right child is red. By making some color changes and performing a left rotation on x.p, we can remove the extra black on x, making it singly black, without violating any of the red-black properties. Setting x to be the root causes the while loop to terminate when it tests the loop condition.

Analysis
What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is \(O(lg n)\), the total cost of the procedure without the call to RB-DELETE-FIXUP takes \(O(lg n)\) time. Within RB-DELETE-FIXUP, each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the while loop can be repeated, and then the pointer x moves up the tree at most \(O(lg n)\) times, performing no rotations. Thus, the procedure RB-DELETE-FIXUP takes \(O(lg n)\) time and performs at most three rotations, and the overall time for RB-DELETE is therefore also \(O(lg n)\).
13.4 Deletion

Figure 13.7  The cases in the while loop of the procedure RB-DELETE-FIXUP. Darkened nodes have color attributes BLACK, heavily shaded nodes have color attributes RED, and lightly shaded nodes have color attributes represented by $c$ and $c'$, which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by $x$ has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes $B$ and $D$ and performing a left rotation. (b) In case 2, the extra black represented by the pointer $x$ moves up the tree by coloring node $D$ red and setting $x$ to point to node $B$. If we enter case 2 through case 1, the while loop terminates because the new node $x$ is red-and-black, and therefore the value $c$ of its color attribute is RED. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes $C$ and $D$ and performing a right rotation. (d) Case 4 removes the extra black represented by $x$ by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.
Exercises

13.4-1
Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

13.4-2
Argue that if in RB-DELETE both \( x \) and \( x.p \) are red, then property 4 is restored by the call to RB-DELETE-FIXUP(\( T, x \)).

13.4-3
In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

13.4-4
In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel \( T.nil \)?

13.4-5
In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees \( \alpha, \beta, \ldots, \zeta \), and verify that each count remains the same after the transformation. When a node has a color attribute \( c \) or \( c' \), use the notation count(\( c \)) or count(\( c' \)) symbolically in your count.

13.4-6
Professors Skelton and Baron are concerned that at the start of case 1 of RB-DELETE-FIXUP, the node \( x.p \) might not be black. If the professors are correct, then lines 5–6 are wrong. Show that \( x.p \) must be black at the start of case 1, so that the professors have nothing to worry about.

13.4-7
Suppose that a node \( x \) is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.
Problems

13-1 Persistent dynamic sets
During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set persistent. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.

Consider a persistent set $S$ with the operations INSERT, DELETE, and SEARCH, which we implement using binary search trees as shown in Figure 13.8(a). We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root $r'$ with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).

Assume that each tree node has the attributes key, left, and right but no parent. (See also Exercise 13.3-6.)

![Figure 13.8](https://www.tutorialsduniya.com)

**Figure 13.8** (a) A binary search tree with keys 2, 3, 4, 7, 8, 10. (b) The persistent binary search tree that results from the insertion of key 5. The most recent version of the set consists of the nodes reachable from the root $r'$, and the previous version consists of the nodes reachable from $r$. Heavily shaded nodes are added when key 5 is inserted.
332 Chapter 13 Red-Black Trees

a. For a general persistent binary search tree, identify the nodes that we need to change to insert a key \( k \) or delete a node \( y \).

b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree \( T \) and a key \( k \) to insert, returns a new persistent tree \( T' \) that is the result of inserting \( k \) into \( T \).

c. If the height of the persistent binary search tree \( T \) is \( h \), what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of new nodes allocated.)

d. Suppose that we had included the parent attribute in each node. In this case, PERSISTENT-TREE-INSERT would need to perform additional copying. Prove that PERSISTENT-TREE-INSERT would then require \( \Omega(n) \) time and space, where \( n \) is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are \( O(\lg n) \) per insertion or deletion.

13-2 Join operation on red-black trees

The join operation takes two dynamic sets \( S_1 \) and \( S_2 \) and an element \( x \) such that for any \( x_1 \in S_1 \) and \( x_2 \in S_2 \), we have \( x_1.key \leq x.key \leq x_2.key \). It returns a set \( S = S_1 \cup \{x\} \cup S_2 \). In this problem, we investigate how to implement the join operation on red-black trees.

a. Given a red-black tree \( T \), let us store its black-height as the new attribute \( T.bh \). Argue that RB-INSERT and RB-DELETE can maintain the \( bh \) attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through \( T \), we can determine the black-height of each node we visit in \( O(1) \) time per node visited.

We wish to implement the operation RB-JOIN(\( T_1, x, T_2 \)), which destroys \( T_1 \) and \( T_2 \) and returns a red-black tree \( T = T_1 \cup \{x\} \cup T_2 \). Let \( n \) be the total number of nodes in \( T_1 \) and \( T_2 \).

b. Assume that \( T_1.bh \geq T_2.bh \). Describe an \( O(\lg n) \)-time algorithm that finds a black node \( y \) in \( T_1 \) with the largest key from among those nodes whose black-height is \( T_2.bh \).

c. Let \( T_y \) be the subtree rooted at \( y \). Describe how \( T_y \cup \{x\} \cup T_2 \) can replace \( T_y \) in \( O(1) \) time without destroying the binary-search-tree property.

d. What color should we make \( x \) so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in \( O(\lg n) \) time.
e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

### 13-3 AVL trees

An **AVL tree** is a binary search tree that is **height balanced**: for each node $x$, the heights of the left and right subtrees of $x$ differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node $x$. As for any other binary search tree $T$, we assume that $T.root$ points to the root node.

a. Prove that an AVL tree with $n$ nodes has height $O(\lg n)$. (*Hint:* Prove that an AVL tree of height $h$ has at least $F_h$ nodes, where $F_h$ is the $h$th Fibonacci number.)

b. To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure $\text{BALANCE}(x)$, which takes a subtree rooted at $x$ whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.\text{right}.h - x.\text{left}.h| \leq 2$, and alters the subtree rooted at $x$ to be height balanced. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure $\text{AVL-INSERT}(x, z)$ that takes a node $x$ within an AVL tree and a newly created node $z$ (whose key has already been filled in), and adds $z$ to the subtree rooted at $x$, maintaining the property that $x$ is the root of an AVL tree. As in $\text{TREE-INSERT}$ from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = \text{NIL}$ and $z.right = \text{NIL}$; also assume that $z.h = 0$. Thus, to insert the node $z$ into the AVL tree $T$, we call $\text{AVL-INSERT}(T.root, z)$.

d. Show that $\text{AVL-INSERT}$, run on an $n$-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

### 13-4 Treaps

If we insert a set of $n$ items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree. What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?
Figure 13.9 A treap. Each node $x$ is labeled with $x.key$, $x.priority$. For example, the root has key $G$ and priority $4$.

We will examine a data structure that answers this question in the affirmative. A **treap** is a binary search tree with a modified way of ordering the nodes. Figure 13.9 shows an example. As usual, each node $x$ in the tree has a key value $x.key$. In addition, we assign $x.priority$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- If $v$ is a left child of $u$, then $v.key < u.key$.
- If $v$ is a right child of $u$, then $v.key > u.key$.
- If $v$ is a child of $u$, then $v.priority > u.priority$.

(This combination of properties is why the tree is called a “treap”: it has features of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, with associated keys, into a treap. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities, i.e., $x_i.priority < x_j.priority$ means that we had inserted $x_i$ before $x_j$.

**a.** Show that given a set of nodes $x_1, x_2, \ldots, x_n$, with associated keys and priorities, all distinct, the treap associated with these nodes is unique.

**b.** Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. Then we call the insertion algorithm, which we call **TREAP-INSERT**, whose operation is illustrated in Figure 13.10.
Problems for Chapter 13

Figure 13.10 The operation of TREAP-INSERT. (a) The original treap, prior to insertion. (b) The treap after inserting a node with key $C$ and priority 25. (c)–(d) Intermediate stages when inserting a node with key $D$ and priority 9. (e) The treap after the insertion of parts (c) and (d) is done. (f) The treap after inserting a node with key $F$ and priority 2.
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Topic</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>String Matching</td>
<td>Ch 11.1 – 11.3</td>
</tr>
</tbody>
</table>

TutorialsDuniya.com

Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Programming in Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- Database Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Computer System Architecture
- Programming in Java
- Discrete Structures
- Data Structures
- Operating Systems
- Computer Networks
- Algorithms
- Android Programming
- Database Management Systems
- PHP Programming
- Software Engineering
- JavaScript
- Theory of Computation
- Java Server Pages
- Operational Research
- Python
- System Programming
- Microprocessor
- Data Mining
- Artificial Intelligence
- Computer Graphics
- Machine Learning
- Data Science

- Compiled Books: https://www.tutorialsduniya.com/compiled-books
- Programs: https://www.tutorialsduniya.com/programs
- Question Papers: https://www.tutorialsduniya.com/question-papers
- Python Notes: https://www.tutorialsduniya.com/python
- Java Notes: https://www.tutorialsduniya.com/java
- JavaScript Notes: https://www.tutorialsduniya.com/javascript
- JSP Notes: https://www.tutorialsduniya.com/jsp
- Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
- OR Notes: https://www.tutorialsduniya.com/operational-research
11

String Matching

11.1 Introduction
11.2 A Straightforward Solution
11.3 The Knuth-Morris-Pratt Algorithm
11.4 The Boyer-Moore Algorithm
11.5 Approximate String Matching
11.1 Introduction

In this chapter we study the problem of detecting the occurrence of a particular substring, called a pattern, in another string, called the text. The problem is usually presented in the context of character strings and arises often in text processing, and we will assume this context in our discussion and examples. However, the solutions presented are applicable to other contexts, such as matching a string of bytes containing graphical data, machine code, or other data, and matching a sublist of a linked list. The first three algorithms described in this chapter look for an exact match of the pattern. The problem of approximate matching is addressed in Section 11.5. We use the following nomenclature throughout the chapter.

**Definition 11.1** Notation for patterns and text

This chapter uses the following notational conventions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>The pattern being searched for</td>
</tr>
<tr>
<td>$T$</td>
<td>The text in which $P$ is sought</td>
</tr>
<tr>
<td>$m$</td>
<td>The length of $P$</td>
</tr>
<tr>
<td>$n$</td>
<td>The length of $T$, not known to the algorithm, used for analysis only</td>
</tr>
<tr>
<td>$p_i$, $t_i$</td>
<td>The $i$th characters in $P$ and $T$ are denoted with lowercase letters and subscripts. The initial index of both $P$ and $T$ is assumed to be 1.</td>
</tr>
<tr>
<td>$j$</td>
<td>Current position within $T$</td>
</tr>
<tr>
<td>$k$</td>
<td>Current position within $P$</td>
</tr>
</tbody>
</table>

We assume a boolean function is given that tells us when we are beyond the last character of the text: `endText(T, j)` returns true if $j$ is greater than the index of the last character of $T$, and returns false otherwise.

For the pseudocode in this chapter we assume that both $P$ and $T$ are arrays of characters. This is a reasonable assumption for $P$ because it is assumed to be of relatively short length, and available for preprocessing by the string-matching algorithms. However, $T$ may well be a different type, may be extremely long, and may not be available all at once in memory. However, we will see that the algorithms perform only limited operations on $T$, and do not use the full flexibility of array access, so they can be adapted easily for applications in which $T$ is not an array. Some of these issues are addressed in the exercises. We will assume that $n$ is fairly large relative to $m$. Through the use of `endText`, algorithms do not need to know $n$. However, $n$ appears in the analysis.

Java sidelong: Java provides a built-in class named `String`, which is not the same as an array of characters. In the interest of language independence we do not use this built-in class.

You should think about the string-matching problem and write out, or at least outline, an algorithm to solve it before proceeding. Your algorithm will probably be very similar to the first one we present here, which is fairly straightforward.
11.2 A Straightforward Solution

Let us first examine a very straightforward procedure for string matching. Starting at the beginning of each string, we compare characters, one after the other, until either the pattern is exhausted or a mismatch is found. In the former case we are done; a copy of the pattern has been found in the text. In the latter case we start over again, comparing the first pattern character with the second text character. In general, when a mismatch is found, we (figuratively) slide the pattern one more place forward over the text and start again comparing the first pattern character with the next text character.

Example 11.1 Straightforward string matching

Comparisons are done (in left-to-right order) on the pairs of characters indicated by arrows. When a mismatch occurs, the pattern is moved one position forward with respect to the text, and the comparisons start again at the left end of the pattern.

\[
P : \quad \begin{array}{ccccccc} A & B & A & B & C \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ T : \quad \begin{array}{ccccccc} A & B & A & B & B & C & A \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \uparrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \end{array}
\]

Observe that moving the pattern all the way past the point where the first mismatch occurred could fail to detect an occurrence of the pattern.

Algorithm 11.1 Simple String Matching

Input: \( P \) and \( T \), the pattern and text strings; \( m \), the length of \( P \). The pattern is assumed to be nonempty.

Output: The return value is the index in \( T \) where a copy of \( P \) begins, or \(-1\) if no match for \( P \) is found.

Remarks: The general picture is shown in Figure 11.1. The index variable \( i \) is not really needed in the algorithm since it can be computed from \( j \) and \( k \) (that is, \( i = j - k + 1 \)). The function \( \text{endText} \) is as defined in Definition 11.1.

For any query, contact us at help@tutorialsduniya.com
Analysis

We will count the character comparisons done by our string-matching algorithms. This is certainly reasonable for Algorithm 11.1, given its simple loop structure. There are few easy cases. If the pattern appears at the beginning of the text, \( m \) comparisons are done. If \( p_1 \) is not in \( T \) at all, \( n \) comparisons are done. What is the worst case? The number of comparisons would be maximized if for each value of \( i \)—that is, each possible starting place for \( P \) in \( T \)—all but the last character of \( P \) matched the corresponding text characters. Thus the number of character comparisons in the worst case is at most \( mn \), and the complexity of the algorithm is in \( O(mn) \).

For some algorithms, inputs that require a lot of work at one step may require very little work at another step. Thus adding up the maximum possible work at each step gives an upper bound but not necessarily an exact value for the work done in the worst case.

To show that the worst case requires (roughly) \( mn \) comparisons (i.e., to show that the worst-case complexity is in \( O(mn) \)), we must show that the situation described can really occur, that is, that \( P \) and \( T \) can be constructed so that all characters in \( P \) but the last one match corresponding characters beginning anywhere in \( T \). Let \( P = 'AA \cdots AB' \) (\( m - 1 \) \( A \)'s followed by a \( B \)) and \( T = 'A \cdots A' \) (\( n \) \( A \)'s).
This worst-case example is not one that occurs often in natural language text. In fact, Algorithm 11.1 works quite well on the average for natural language. In some empirical studies the algorithm did only about 1.1 character comparisons for each character in T (up to the point where a match was found or to the end of T’ if no match was found). Thus few characters in the text had to be examined more than once.

Algorithm 11.1 has a property that in some applications is undesirable: It may often be necessary to back up in the text string (by the amount backup – 1, since backup is subtracted from 1, and 1 is added to initiate a new match attempt in the while loop). If the text is being read from an input source that does not permit backing up, this makes the algorithm difficult to use (see Exercise 11.4 for a space-efficient solution). The algorithm we present in the next section was devised specifically to eliminate the need to back up in the text. It turned out to be faster (in the worst case) as well.

11.3 The Knuth-Morris-Pratt Algorithm

We first describe briefly, without formal algorithms, an approach to the pattern-matching problem that has some important good points but also some drawbacks. The construction used by the main algorithm of this section was suggested by the method we describe now and salvages some of its advantages while eliminating the disadvantages.

11.3.1 Pattern Matching with Finite Automata

Given a pattern P, it is possible to construct a finite automaton that can be used to scan the text for a copy of P very quickly. A finite automaton can easily be interpreted as a special kind of machine or flowchart, and a knowledge of automata theory is not necessary to understand this method.

Definition 11.2

Let Σ be the alphabet, or set of characters, from which the characters in P and T may be chosen, and let α = |Σ|. The flowchart, or finite automaton, has two types of nodes:

1. Some read nodes, which mean “Read the next text character. If there are no further characters in the text string, halt; there is no match.” One read node is designated the start node.

2. A stop node, which means “Stop; a match was found.” It is marked with a •. ■

The flowchart has α arrows leading out from each read node. Each arrow is labeled with a character from Σ. The arrow that matches the text character just read is the arrow to be followed; that is, it indicates which node to go to next. You should study the example in Figure 11.2 to understand why the arrows point where they do. The read nodes serve as a sort of memory. For instance, if execution reaches the third read node, the last two characters read from the text were A’s. What preceded them is irrelevant. For a successful match, they must be followed immediately by a B and a C. If the next character is a B we can move on to node 4, which remembers that AAB has appeared. On the other hand, if
the next character read at node 3 were a C, we would have to return to node 1 and wait for another A to begin the pattern.

Once the flowchart for the pattern is constructed, the text can be tested for an occurrence of the pattern by examining each text character only once, hence in $O(n)$ time. This is a big improvement over Algorithm 11.1, both in timing and in the fact that, once a text character has been examined, it never has to be reconsidered: there is no backing up in the text. The difficulty is constructing the finite automaton—that is, deciding where all the arrows go. There are well-known algorithms to construct the finite automaton to recognize a particular pattern, but in the worst case these algorithms require a lot of time. The difficulty arises from the fact that there is an arrow for each character in $\Sigma$ leading out from each read node. It takes time to determine where each arrow should point, and space to represent more arrows. Thus a better algorithm will have to eliminate some of the arrows.

11.3.2 The Knuth-Morris-Pratt Flowchart

When constructing the finite automaton for a pattern $P$, it is easy to put in the arrows that correspond to a successful match. For example, when drawing Figure 11.2 for the pattern ‘AABC’, the first step is to draw

The difficult part is the insertion of the rest of the arrows. The Knuth-Morris-Pratt algorithm (which, for brevity, will be called the KMP algorithm) also constructs a sort of flowchart to be used to scan the text. The KMP flowchart contains the easy arrows—that is, the ones to follow if the desired character is read from the text—but it contains only one other arrow from each node, an arrow to be followed if the desired character was not read from the text. The arrows are called the success links and the failure links, respectively. The KMP flowchart differs from the finite automaton in several details: The character labels of the KMP flowchart are on the nodes rather than on the arrows; the next character from the text is read only after a success link has been followed; the same text character is reconsidered if a failure link is followed; there is an extra node that causes a new text character to be read. The scan starts at this node. As in the finite automaton, if the node labeled with the * is reached, a copy of the pattern has been found; if the end of the text is reached elsewhere in the flowchart, the scan terminates unsuccessfully. This
11.3 The Knuth-Morris-Pratt Algorithm

The KMP flowchart for $P = \text{'ABABCB'}$

Get next text character

<table>
<thead>
<tr>
<th>KMP cell number</th>
<th>Text being scanned</th>
<th>Success (+) or Failure (-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>more failure</td>
</tr>
</tbody>
</table>

Table 11.1 Action of the KMP flowchart in Figure 11.3 for the pattern 'ABABCB' on the text 'ACABAABABA'

An informal description of the scanning procedure should enable you to use the KMP flowchart in Figure 11.3 to scan a text string. Try 'ACABAABABA' and refer to Table 11.1 if you have difficulty.

We now need a computer representation of the KMP flowchart, an algorithm to construct it (to determine how to set the failure links), a formal algorithm for the scan procedure, and an analysis of the two algorithms.
11.3.3 Construction of the KMP Flowchart

The flowchart representation is quite simple, it uses two arrays, one containing the characters of the pattern and one containing the failure links. The success links are implicit in the ordering of the array entries.

Let fail be the array of failure links; fail[k] will be the index of the node pointed to by the failure link at the kth node, for 1 \leq k \leq m. The special node that merely forces the next text character to be read is considered to be the zero-th node; fail[1] = 0. To see how to set the other failure links, we consider an example.

Example 11.2 Setting fail links for the KMP algorithm

Let \textit{P} = 'ABABABCB' and suppose that the first six characters have matched six consecutive text characters as indicated:

\[
\begin{align*}
\text{T :} & \quad . . . A B A B A B x . . . \\
\text{P :} & \quad A B A B A B C B \\
& \quad \downarrow \downarrow \downarrow \downarrow \downarrow \\
\end{align*}
\]

Suppose that the next text character, \textit{x}, is not a 'C'. The next possible place where the pattern could begin in the text is at the third position shown, that is, as follows:

\[
\begin{align*}
\text{T :} & \quad . . . A B A B A B C B \\
\text{P :} & \quad A B A B A B C B \\
\end{align*}
\]

The pattern is moved forward so that the longest initial segment that matches part of the text preceding \textit{x} is lined up with that part of the text. Now \textit{x} should be tested to see if it is an A to match the third A of the pattern. Thus the failure link for the node containing the C should point to the node containing the third A. ■

The general picture is shown in Figure 11.4. When a mismatch occurs, we want to slide \textit{P} forward, but maintain the longest overlap of a prefix of \textit{P} with a suffix of the part of the text that has matched the pattern so far. Thus the current text character should be compared to \textit{p_r next}; that is, fail[k] should be \textit{r}. But we want to construct the flowchart before we ever see \textit{T}. How do we determine \textit{r} without knowing \textit{T}? The key observation is that when we do scan \textit{T}, the part of \textit{T} just scanned will have matched the part of \textit{P} just scanned, so we just need to find the longest overlap of a prefix of \textit{P} with a suffix of the part of \textit{P} just scanned.

Definition 11.3 Fail links

We define fail[k] as the largest \textit{r} (with \textit{r} < \textit{k}) such that \textit{p_1 \ldots p_r = \ldots p_{k-1}} matches \textit{p_k \ldots p_{k-1}}. That is, the \textit{(r - 1)-character prefix} of \textit{P} is identical to the \textit{(r - 1)-character substring ending at index k - 1}. Thus the fail links are determined by repetition within \textit{P} itself. ■

For any query, contact us at help@tutorialsduniya.com
11.3 The Knuth-Morris-Pratt Algorithm

(a) Mismatch at \( p_k \) and \( t_j \)

(b) Slide \( p \) to line up the longest prefix that matches a suffix of the scanned characters.

Figure 11.4 Sliding the pattern for the KMP algorithm

An occurrence of the pattern could be missed if \( r \) were chosen too small. (Consider what would happen if in Example 11.2 the failure link for \( C \) was set to point to the second \( A \), and if \( s = A \) and is followed by \( BCB \) in the text.)

Although we have described the correct values for the failure links, we still don't have an algorithm for efficiently computing them. We can define fail recursively. Suppose that the first \( k - 1 \) failure links have been computed. Then we have the picture in Figure 11.5(a). To assign \( \text{fail}[k] \) we need to match a substring of \( P \) ending at \( k - 1 \). To simplify the notation, we will let \( s = \text{fail}[k - 1] \). The easy case is when \( p_{k - 1} = p_s \), because we already know that \( p_1 \ldots p_{s - 1} \) matches the \( (s - 1) \)-character substring ending at \( k - 2 \). Then the two matching sequences in Figure 11.5(a) can be extended by one more character, so in this case \( \text{fail}[k] \) is assigned \( s + 1 \).

Example 11.3 Computing KMP fail links—a simple case

In Figure 11.6, \( \text{fail}[6] = 4 \) because \( p_1 p_2 p_3 \) matches \( p_3 p_4 p_5 \). Since \( p_6 = p_4 \), \( \text{fail}[7] \) is assigned 5. This tells us that \( p_1 \ldots p_4 \) matches the four-character substring ending at index 6.

What if \( p_{k - 1} \neq p_s \)? We must find a prefix of \( P \) that matches a substring ending at \( k - 1 \). In this case the match in Figure 11.5(a) cannot be extended, so we look farther back. Let \( s_2 = \text{fail}[s] \). By the properties of the failure links we have the matches shown in

For any query, contact us at help@tutorialsduniya.com
Chapter 11  String Matching

Figure 11.5  Computing fail links: Index $s$ is equal to fail[$k$−1].

Figure 11.6  Computing fail links: Dashed edges are discussed in Examples 11.3 and 11.4.

Example 11.4  Computing KMP fail links—the recursive case

Again look at Figure 11.6. To compute fail[8], $s = \text{fail}[7] = 5$. But $p_7 \neq p_5$, so recompute $s = \text{fail}[5] = 3$. But $p_7 \neq p_3$ either, so recompute $s = \text{fail}[3] = 1$. Still $p_7 \neq p_1$. Finally, $s = \text{fail}[1] = 0$ ends the search, and fail[8] is assigned $s + 1 = 1$.

For any query, contact us at help@tutorialsduniya.com
Algorithm 11.2 KMP Flowchart Construction

**Input:** $P$, a string of characters; $m$, the length of $P$.

**Output:** fail, the array of failure links, defined for indexes $1, \ldots, m$. The array is passed in and the algorithm fills it.

```c
void kmpSetup(char* P, int m, int* fail)
{
    int k, s;
    fail[1] = 0;
    for (k = 2; k <= m; k++)
    {
        s = fail[k-1];
        while (s >= 1)
        {
            if (P[k] == P[s])
            {
                break;
            }
            s = fail[s];
        }
        fail[k] = s + 1;
    }
}
```

### 11.3.4 Analysis of KMP Flowchart Construction

Let $m$ be the length of the pattern, $P$. For this discussion we assume $m \geq 2$. It is easy to see that the complexity of Algorithm 11.2 is in $O(m^2)$. The body of the for loop is executed $m - 1$ times, and each time, the body of the while loop is executed at most $m$ times because $s$ starts somewhere in $P$ and "jumps" backward, at worst to zero. But this analysis is not careful enough.

We will count character comparisons, as we did for Algorithm 11.1. Since the character comparison is executed in each pass through the while loop, the running time of the algorithm is bounded by a multiple of the number of character comparisons. (Actually, since the character comparison is not executed when $s = 0$, we should also note that the condition $s = 0$ cannot occur more than $m - 1$ times.)

We call a comparison **successful** if $p_k = p_{k-1}$ and **unsuccessful** otherwise. A successful comparison breaks out of the while loop so at most $m - 1$ successful comparisons are done (one for each $k$ from 2 through $m$). After every unsuccessful comparison $s$ is decreased (since fail[$s$] < $s$), so we can bound the number of unsuccessful comparisons by determining how many times $s$ can decrease. Observe the following:

1. $s$ is initially assigned 0, when $k = 2$.
2. $s$ is increased only by executing line 8 on one pass of the for loop, followed by line 3 on the subsequent pass; these two statements increase $s$ by 1. This occurs $m - 2$ times.
3. $s$ is never negative.

Therefore $s$ cannot be decreased more than $m - 2$ times. Thus the number of unsuccessful comparisons is at most $m - 2$ and the total number of character comparisons is at most $2m - 3$. Observe that, to count character comparisons, we actually counted the number of times the index $s$ changed. The latter is another good measure of the work done by the algorithm. The important conclusion is that the complexity of the construction of the flowchart is linear in the length of the pattern.
11.3.5 The Knuth-Morris-Pratt Scan Algorithm

We have already informally described the procedure for using the KMP flowchart to scan the text. The algorithm follows.

Algorithm 11.3 KMP Scan

**Input:** $P$ and $T$, the pattern and text strings; $m$, the length of $P$; $fail$, the array of failure links set up in Algorithm 11.2. The length of $P$ would have been found when setting up the fail array. The pattern is assumed to be nonempty.

**Output:** The return value is the index in $T$ where a copy of $P$ begins, or $−1$ if no match for $P$ is found.

**Remark:** The function endText is as defined in Definition 11.1.

```c
int kmpScan(char[] P, char[] T, int m, int[] fail)
    int match;
    int j, k;
    // j indexes text characters;
    // k indexes the pattern and fail array.
    match = −1;
    j = 1; k = 1;
    while (endText(T, j) == false)
        if (k > m)
            match = j − m; // Match found
            break;
        if (k == 0)
            j ++;
            k = 1; // Start pattern over
        else if (Tj == Pk)
            j ++;
            k ++;
        else
            // Follow fail arrow.
            k = fail[k];
            // Continue loop.
            return match;
```

The analysis of the scan algorithm uses an argument very similar to that used to analyze the algorithm to set up the failure links, and it is left as Exercise 11.8. The number of character comparisons done by Algorithm 11.3 is at most $2n$, where $n$ is the length of the text, $T$. Thus the Knuth-Morris-Pratt pattern-matching algorithm, which is comprised of Algorithms 11.2 and 11.3, does $Θ(n + m)$ operations in the worst case, a significant improvement over the $Θ(nm)$ worst-case complexity of Algorithm 11.1. Some empirical
studies have shown that the two algorithms do roughly the same number of character comparisons on the average (for natural language text), but the KMP algorithm never has to back up in the text.
Exercises

Section 11.2 A Straightforward Solution

11.1 Rewrite Algorithm 11.1 eliminating the variable $i$.

11.2 Rewrite Algorithm 11.1 to work on inputs that are linked lists. For simplicity, assume the element type is int. Use the InList abstract data type operations of Section 2.3.2, and assume that $T$ and $P$ are objects in this class.

11.3 In this exercise you will design operations for a Text abstract data type that will do what simpleScan needs to do with $T$, without having to assume that $T$ is an array. Try to make them general enough that other string-matching algorithms can probably use...
them. We have already assumed endText was available. Some other suggested names are advanceText, backupText, and getChar.

a. Write the specifications, but do not implement the operations.

b. Show how to modify simpleScan to use your operations on T.

11.4 Suppose the text T is too long to store all at once in memory, so it is read in as needed. Algorithm 11.1 may need about m previous characters of T, to the left of f_j. That is, it may need to back up about m positions.

In this exercise you will design a modification of the Queue ADT, called OpenQueue, which keeps elements (characters in this case) in FIFO order, as does a regular queue, but permits access to any element in the queue, not just the front element.

a. Write the specifications for your new operations.

b. Show how to implement the needed operations efficiently using an array, using suggestions from Exercise 2.16.

c. Outline how to implement the Text ADT of Exercise 11.3 using the OpenQueue ADT.

The idea is to keep enough characters in the queue to accommodate any backing up the algorithm may need to do. Assume the algorithm knows the maximum amount it might ever need to back up at the time the OpenQueue object is created.

Section 11.3 The Knuth-Morris-Pratt Algorithm

11.5 Draw the finite automaton (flowchart) for the pattern ‘ABAAAB’, where \( \Sigma = \{A, B, C\} \).

11.6 Give the fail indexes used by the KMP algorithm for the following patterns:

a. AAAB

b. AABACAABABA

c. ABRACADABA

d. ASTRACASTRA

11.7 Give a pattern beginning with an A and using only letters from \{A, B, C\} that would have the following fail indexes (for the KMP algorithm):

0 1 1 2 3 4 2 2

11.8 Show that kmpScan (Algorithm 11.3) does at most 2n character comparisons.

11.9 How will the KMP algorithms behave if the pattern and/or the text are null (have length zero)? Will they "crash"? If not, will their output be meaningful and correct?

11.10 Recall that the pattern \( P = A \cdots AB \) (m − 1 A's followed by one B) and the text string \( T = A \cdots A \) (n A's) are a worst-case input for Algorithm 11.1.

a. Give the values of the fail indexes for \( P \). Exactly how many character comparisons are done by Algorithm 11.2 to compute them?
Chapter 11  String Matching

b. Exactly how many character comparisons are done by kmpScan to scan $T$ for an occurrence of $P$?

c. Given an arbitrarily large $m$, find a pattern $Q$ with $m$ letters such that kmpSetup does more character comparisons for $Q$ than it does for the pattern $P$ with $m$ letters described above.

11.11 Prove that Algorithm 11.2 sets the KMP failure links so that $\text{fail}[k]$ is the largest $r$ (with $r < k$) such that $p_1 \cdots p_{r-1}$ matches $p_{k-r+1} \cdots p_{k-1}$.

11.12 The strategy for setting the fail links for the KMP algorithm has a flaw that is illustrated by Figure 11.3. If a mismatch occurs at the fourth character, a $B$, $\text{fail}[4]$ points us back to another $B$, which of course will not match the current text character either. Modify Algorithm 11.2 so that fail values satisfy the condition stated in Section 11.3.3 (and repeated in the previous exercise) and also the condition that $p_r \neq p_k$. (Be careful; a common first guess does not work.)

11.13 Rewrite the KMP algorithms to work on inputs that are linked lists. For simplicity, assume the element type is int. Use the IntList abstract data type operations of Section 2.3.2, and assume that $T$ and $P$ are objects in this class.

11.14 How would you modify kmpScan (Algorithm 11.3) to read the text from input, one character at a time, instead of accessing string $T$? Assume the function read() returns an int, which equals the next input character, unless end of file has been reached, in which case it returns $-1$. Do you need the full capabilities of the Text ADT proposed in Exercise 11.3? Explain why or why not.

Section 11.4  The Boyer-Moore Algorithm

11.15 List the values in the charjump array for the Boyer-Moore algorithm for the following patterns assuming that the alphabet is $\{A, B, \ldots, Z\}$.

a. ABRACADABRA
b. ASTRACASTRA

c. ABBACABABA

d. ABRACADABRA
d. ASTRACASTRA

d. ASTRACASTRA

d. ASTRACASTRA

d. ASTRACASTRA

11.17 As Example 11.5 showed, just using the charjump values, without using matchjump, can give a very fast scan. However, the statement

$$j \leftarrow \max(\text{charjump}[j], \text{matchjump}[k])$$

For any query, contact us at help@tutorialsduniya.com
in Algorithm 11.6 cannot simply be replaced by

\[ j += \text{charJump}[j]. \]

Why not? What other (small) change is needed to make the scan algorithm work?

11.18 Recall that the pattern \( P = 'A \cdots AB' (m - 1 \text{ A's followed by one B}) \) and the text string \( T = 'A \cdots A' (m \text{ A's}) \) are a worst-case input for Algorithm 11.1.

a. Give the values of the charJump, sufX, and matchJump arrays for \( P \) assuming that the alphabet is \( \{A, B, \ldots, Z\} \).

b. Exactly how many character comparisons are done by \texttt{boyerMooreScan} to scan \( T \) for an occurrence of \( P \)?

11.19 Suppose the text is being read as needed, one character at a time. Give a formula relating \( k, m, \) and \text{matchJump}[k] to the number of new text characters needed when there is a mismatch at \( p_k \).

11.20 Suppose \( P \) and \( T \) are bitstrings.

a. Show the values in the charJump, sufX, and matchJump arrays for the pattern 
\[ 101010101. \]

b. For bitstrings in general, which array, charJump or matchJump, will yield the longer “jumps”?

Section 11.5 Approximate String Matching

11.21 An algorithm for finding an exact string match need tell us only where the pattern begins in the text or where it ends in the text. We can determine the unspecified end of the match in the text because we know the length of the pattern. This is not the case with approximate matching because we don’t know how many characters are missing from the pattern or the text. Show how to modify or extend the algorithm for detecting \( k \)-approximate matches so that it tells where the approximate match of the pattern begins in \( T \).

11.22

a. Write out the algorithm for \( k \)-approximate matching. How much space does it require?

b. Show how to use a version of the \texttt{OpenQueue ADT}, introduced in Exercise 11.4, to avoid using an amount of space that depends on \( n. \text{ Hint:} \) Let the elements stored in the open queue be arrays of \( m + 1 \) integers, corresponding to the columns of the table \( D \). What is the maximum number of columns that needs to be available at any one time?

Additional Problems

11.23 Rewrite each of the three scan algorithms (Algorithms 11.1, 11.3, and 11.6) so that they find all occurrences of the pattern in the text.

For any query, contact us at help@tutorialsduniya.com
11.24 $P$ is a character string (of length $n$) consisting of letters and at most one asterisk ("*"). The asterisk is a "wild-card" character; it can match any sequence of zero or more characters. For example, if $P = \text{'sunday'}$ and $T = \text{'happy\_sunday\_monday'}$, there is a match beginning at the ‘s’ and ending at the last ‘y’; the asterisk “matches” daemon. Give an algorithm to find a match of $P$ in a text string $T$ (consisting of $n$ characters), if there is one, and give an upper bound on the order of its worst-case time.

11.25 Let $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_n$ be two character strings. We say that $X$ is a cyclic shift of $Y$ if there is some $r$ such that $X = y_{r+1} \ldots y_n y_1 \ldots y_r$. Give an $O(n)$ algorithm to determine if $X$ is a cyclic shift of $Y$.

11.26

a. Write an efficient algorithm to determine if a (long) string of text contains 25 consecutive blanks. (Do not just give an exact copy of an algorithm in the text; customize it.)

b. Construct a worst-case (or near worst-case) example for your algorithm. How many character comparisons are done in this case?

c. Suppose the text string contains ordinary English text where blanks separate words and sentences, but there is very rarely more than one blank together. If the text length is $n$, approximately how many character comparisons will your algorithm do?

11.27 Investigate the problem of finding any one of a finite set of patterns in a text string. Can you extend any of the algorithms in this chapter to produce an algorithm that does better than scan for each of the patterns separately?

Programs

1. Implement all three exact string searching algorithms, including a counter for the number of character comparisons done; run a large set of test cases; and compare the results. Use the techniques of Exercises 11.3 and 11.4 to manage backing up and jumping forward in text, so that you don’t need to store the entire text in memory.

2. Write a program for the $k$-approximate matching algorithm, storing at most two columns at a time. Include the enhancements for Exercise 11.21.

Notes and References

Crochemore and Rytter (1994) is a book on text algorithms in general. It includes the Knuth-Morris-Pratt and Boyer-Moore algorithms and $k$-approximate matching. The main references for the algorithms presented here are Knuth, Morris, and Pratt (1977) and Boyer and Moore (1977). The first phase of Algorithm 11.5 is based on Knuth, Morris, and Pratt (1977) and the second phase is based on an idea attributed to K. Mehlhorn by Smit (1982), Guibas and Odlyzko (1977), Galil (1979), and Apostolico and Giancarlo (1986).
Get FREE Compiled Books, Notes, Programs, Books, Question Papers with Solution* etc of following subjects from https://www.tutorialsduniya.com.

- C and C++
- Computer System Architecture
- Programming in Java
- Discrete Structures
- Data Structures
- Operating Systems
- Computer Networks
- Algorithms
- Android Programming
- Database Management Systems
- PHP Programming
- Software Engineering
- JavaScript
- Theory of Computation
- Java Server Pages
- Operational Research
- Python
- System Programming
- Microprocessor
- Data Mining
- Artificial Intelligence
- Computer Graphics
- Machine Learning
- Data Science

 Compiled Books: https://www.tutorialsduniya.com/compiled-books
 Programs: https://www.tutorialsduniya.com/programs
 Question Papers: https://www.tutorialsduniya.com/question-papers
 Python Notes: https://www.tutorialsduniya.com/python
 Java Notes: https://www.tutorialsduniya.com/java
 JavaScript Notes: https://www.tutorialsduniya.com/javascript
 JSP Notes: https://www.tutorialsduniya.com/jsp
 Microprocessor Notes: https://www.tutorialsduniya.com/microprocessor
 OR Notes: https://www.tutorialsduniya.com/operational-research