# *ALGORITHMS*

## DESIGN TECHNIQUES AND ANALYSIS

M. H. Alsuwaiyel

Information & Computer Science Department

KFUPM

July, 1999

**Algorithm 1.13** FIRST
**Input:** A positive integer $n$ and an array $A[1..n]$ with $A[j] = j, 1 \leq j \leq n$.
**Output:** $\sum_{j=1}^{n} A[j]$.

```
1. $sum \leftarrow 0$
2. for $j \leftarrow 1$ to $n$
3.     $sum \leftarrow sum + A[j]$
4. end for
5. return $sum$
```

**Algorithm 1.14** SECOND
**Input:** A positive integer $n$.
**Output:** $\sum_{j=1}^{n} j$.

```
1. $sum \leftarrow 0$
2. for $j \leftarrow 1$ to $n$
3.     $sum \leftarrow sum + j$
4. end for
5. return $sum$
```

elementary operations performed by both algorithms is the same.

## 1.15 Exercises

**1.1.** Let $A[1..60] = 11, 12, \ldots, 70$. How many comparisons are performed by Algorithm BINARYSEARCH when searching for the following values of $x$?
(a) 33. (b) 7. (c) 70. (d) 77.

**1.2.** Let $A[1..2000] = 1, 2, \ldots, 2000$. How many comparisons are performed by Algorithm BINARYSEARCH when searching for the following values of $x$?
(a) $-3$. (b) 1. (c) 1000. (d) 4000.

**1.3.** Draw the decision tree for the binary search algorithm with an input of
(a) 12 elements. (b) 17 elements. (c) 25 elements. (d) 35 elements.

**1.4.** Illustrate the operation of Algorithm SELECTIONSORT on the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 45 | 33 | 24 | 45 | 12 | 12 | 24 | 12 |

How many comparisons are performed by the algorithm?

**1.5.** Consider modifying Algorithm SELECTIONSORT as shown in Algorithm

MODSELECTIONSORT.

**Algorithm 1.15** MODSELECTIONSORT
**Input:** An array $A[1..n]$ of $n$ elements.

**Output:** $A[1..n]$ sorted in nondecreasing order.

```
1. for $i \leftarrow 1$ to $n-1$
2.     for $j \leftarrow i+1$ to $n$
3.         if $A[j] < A[i]$ then interchange $A[i]$ and $A[j]$
4.     end for
5. end for
```

(a) What is the minimum number of element assignments performed by Algorithm MODSELECTIONSORT? When is this minimum achieved?

(b) What is the maximum number of element assignments performed by Algorithm MODSELECTIONSORT? Note that each interchange is implemented using three element assignments. When is this maximum achieved?

**1.6.** Illustrate the operation of Algorithm INSERTIONSORT on the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 30 | 12 | 13 | 13 | 44 | 12 | 25 | 13 |

.

How many comparisons are performed by the algorithm?

**1.7.** How many comparisons are performed by Algorithm INSERTIONSORT when presented with the input

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 12 | 5 | 6 | 7 | 2 | 9 |

?

**1.8.** Prove Observation 1.4.

**1.9.** Which algorithm is more efficient: Algorithm INSERTIONSORT or Algorithm SELECTIONSORT? What if the input array consists of very large records? Explain.

**1.10.** Illustrate the operation of Algorithm BOTTOMUPSORT on the array

$A[1..16] =$

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 12 | 1 | 5 | 15 | 3 | 4 | 10 | 7 | 2 | 16 | 9 | 8 | 14 | 13 | 6 |

.

How many comparisons are performed by the algorithm?

**1.11.** Illustrate the operation of Algorithm BOTTOMUPSORT on the array

$A[1..11] =$

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 17 | 19 | 5 | 13 | 11 | 4 | 8 | 15 | 12 | 7 |

.

How many comparisons are performed by the algorithm?

**1.12.** Give an array $A[1..8]$ of integers on which Algorithm BOTTOMUPSORT performs

(a) the minimum number of element comparisons.

(b) the maximum number of element comparisons.

**1.13.** Fill in the blanks with either *true* or *false*:

| $f(n)$ | $g(n)$ | $f = O(g)$ | $f = \Omega(g)$ | $f = \Theta(g)$ |
|---|---|---|---|---|
| $2n^3 + 3n$ | $100n^2 + 2n + 100$ | | | |
| $50n + \log n$ | $10n + \log\log n$ | | | |
| $50n \log n$ | $10n \log\log n$ | | | |
| $\log n$ | $\log^2 n$ | | | |
| $n!$ | $5^n$ | | | |

**1.14.** Express the following functions in terms of the $\Theta$-notation.

(a) $2n + 3\log^{100} n$.

(b) $7n^3 + 1000n \log n + 3n$.

(c) $3n^{1.5} + (\sqrt{n})^3 \log n$.

(d) $2^n + 100^n + n!$.

**1.15.** Express the following functions in terms of the $\Theta$-notation.

(a) $18n^3 + \log n^8$.

(b) $(n^3 + n)/(n + 5)$.

(c) $\log^2 n + \sqrt{n} + \log\log n$.

(d) $n!/2^n + n^{n/2}$.

**1.16.** Consider the sorting algorithm shown below, which is called BUBBLESORT.

**Algorithm 1.16** BUBBLESORT
**Input:** An array $A[1..n]$ of $n$ elements.

**Output:** $A[1..n]$ sorted in nondecreasing order.

```
 1. $i \leftarrow 1$;    $sorted \leftarrow false$
 2. while $i \le n - 1$ and not $sorted$
 3.     $sorted \leftarrow true$
 4.     for $j \leftarrow n$ downto $i + 1$
 5.         if $A[j] < A[j-1]$ then
 6.             interchange $A[j]$ and $A[j-1]$
 7.             $sorted \leftarrow false$
 8.         end if
 9.     end for
10.     $i \leftarrow i + 1$
11. end while
```

(a) What is the minimum number of element comparisons performed by the algorithm? When is this minimum achieved?
(b) What is the maximum number of element comparisons performed by the algorithm? When is this maximum achieved?
(c) What is the minimum number of element assignments performed by the algorithm? When is this minimum achieved?
(d) What is the maximum number of element assignments performed by the algorithm? When is this maximum achieved?
(e) Express the running time of Algorithm BUBBLESORT in terms of the $O$ and $\Omega$ notations.
(f) Can the running time of the algorithm be expressed in terms of the $\Theta$-notation? Explain.

**1.17.** Find two monotonically increasing functions $f(n)$ and $g(n)$ such that $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$.

**1.18.** Is $x = O(x \sin x)$? Use the definition of the $O$-notation to prove your answer.

**1.19.** Prove that $\sum_{j=1}^{n} j^k$ is $O(n^{k+1})$ and $\Omega(n^{k+1})$, where $k$ is a positive integer. Conclude that it is $\Theta(n^{k+1})$.

**1.20.** Let $f(n) = \{1/n + 1/n^2 + 1/n^3 + \ldots\}$. Express $f(n)$ in terms of the $\Theta$-notation. (Hint: Find a recursive definition of $f(n)$).

**1.21.** Show that $n^{100} = O(2^n)$, but $2^n \neq O(n^{100})$.

**1.22.** Show that $2^n$ is not $\Theta(3^n)$.

**1.23.** Is $n! = \Theta(n^n)$? Prove your answer.

**1.24.** Is $2^{n^2} = \Theta(2^{n^3})$? Prove your answer.

**1.25.** Carefully explain the difference between $O(1)$ and $\Theta(1)$.

**1.26.** Is the function $\lfloor \log n \rfloor!$ $O(n)$, $\Omega(n)$, $\Theta(n)$? Prove your answer.

**1.27.** Can we use the $\prec$ relation described in Sec. 1.8.6 to compare the order of growth of $n^2$ and $100n^2$? Explain.

**1.28.** Use the $\prec$ relation to order the following functions by growth rate: $n^{1/100}$, $\sqrt{n}$, $\log n^{100}$, $n \log n$, 5, $\log \log n$, $\log^2 n$, $(\sqrt{n})^n$, $(1/2)^n$, $2^{n^2}$, $n!$.

**1.29.** Consider the following problem. Given an array $A[1..n]$ of integers, test each element $a$ in $A$ to see whether it is even or odd. If $a$ is even, then leave it; otherwise multiply it by 2.

(a) Which one of the $O$ and $\Theta$ notations is more appropriate to measure the number of multiplications? Explain.
(b) Which one of the $O$ and $\Theta$ notations is more appropriate to measure the number of element tests? Explain.

**1.30.** Give a more efficient algorithm than the one given in Example 1.25. What is the time complexity of your algorithm?

**1.31.** Consider Algorithm COUNT4 whose input is a positive integer $n$.

**Algorithm 1.17** COUNT4

```
1. comment: Exercise 1.31
2. $count \leftarrow 0$
3. for $i \leftarrow 1$ to $\lfloor \log n \rfloor$
4.     for $j \leftarrow i$ to $i + 5$
5.         for $k \leftarrow 1$ to $i^2$
6.             $count \leftarrow count + 1$
7.         end for
8.     end for
9. end for
```

(a) How many times Step 6 is executed?

(b) Which one of the $O$ and $\Theta$ notations is more appropriate to express the time complexity of the algorithm? Explain.

(c) What is the time complexity of the algorithm?

**1.32.** Consider Algorithm COUNT5 whose input is a positive integer $n$.

**Algorithm 1.18** COUNT5

```
1. comment: Exercise 1.32
2. $count \leftarrow 0$
3. for $i \leftarrow 1$ to $n$
4.     $j \leftarrow \lfloor n/2 \rfloor$
5.     while $j \geq 1$
6.         $count \leftarrow count + 1$
7.         if $j$ is odd then $j \leftarrow 0$  else $j \leftarrow j/2$
8.     end while
9. end for
```

(a) What is the maximum number of times Step 6 is executed when $n$ is a power of 2?

(b) What is the maximum number of times Step 6 is executed when $n$ is a power of 3?

(c) What is the time complexity of the algorithm expressed in the $O$-notation?

(d) What is the time complexity of the algorithm expressed in the $\Omega$-notation?

(e) Which one of the $O$ and $\Theta$ notations is more appropriate to express the time complexity of the algorithm?

**1.33.** Consider Algorithm COUNT6 whose input is a positive integer $n$.

**Algorithm 1.19** COUNT6

```
 1. comment: Exercise 1.33
 2. $count \leftarrow 0$
 3. for $i \leftarrow 1$ to $n$
 4.     $j \leftarrow \lfloor n/3 \rfloor$
 5.     while $j \geq 1$
 6.         for $k \leftarrow 1$ to $i$
 7.             $count \leftarrow count + 1$
 8.         end for
 9.         if $j$ is even then $j \leftarrow 0$ else $j \leftarrow \lfloor j/3 \rfloor$
10.     end while
11. end for
```

(a) What is the maximum number of times Step 7 is executed when $n$ is a power of 2?

(b) What is the maximum number of times Step 7 is executed when $n$ is a power of 3?

(c) What is the time complexity of the algorithm expressed in the $O$-notation?

(d) What is the time complexity of the algorithm expressed in the $\Omega$-notation?

(e) Which one of the $O$ and $\Theta$ notations is more appropriate to express the time complexity of the algorithm?

**1.34.** Write an algorithm to find the maximum and minimum of a sequence of $n$ integers stored in array $A[1..n]$ such that its time complexity is

(a) $O(n)$.

(b) $\Omega(n \log n)$.

**1.35.** Let $A[1..n]$ be an array of integers, where $n > 2$. Give an $O(1)$ time algorithm to find an element in $A$ that is neither the maximum nor the minimum.

**1.36.** Consider the element uniqueness problem: Given a set of integers, determine whether two of them are equal. Give an *efficient* algorithm to solve this problem. Assume that the integers are stored in array $A[1..n]$. What is the time complexity of your algorithm?

**1.37.** Give an algorithm that evaluates an input polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

for a given value of $x$ in time

(a) $\Omega(n^2)$.
(b) $O(n)$.

**1.38.** Let $S$ be a set of $n$ positive integers, where $n$ is even. Give an efficient algorithm to partition $S$ into two subsets $S_1$ and $S_2$ of $n/2$ elements each with the property that the difference between the sum of the elements in $S_1$ and the sum of the elements in $S_2$ is maximum. What is the time complexity of your algorithm?

**1.39.** Suppose we change the word "maximum" to "minimum" in Exercise 1.38. Give an algorithm to solve the modified problem. Compare the time complexity of your algorithm with that obtained in Exercise 1.38.

**1.40.** Let $m$ and $n$ be two positive integers. The *greatest common divisor* of $m$ and $n$, denoted by $gcd(m, n)$, is the largest integer that divides both $m$ and $n$. For example $gcd(12, 18) = 6$. Consider Algorithm EUCLID shown below, to compute $gcd(m, n)$.

**Algorithm 1.20** EUCLID
**Input:** Two positive integers $m$ and $n$.
**Output:** $gcd(m, n)$.

```
1. comment: Exercise 1.40
2. repeat
3.     $r \leftarrow n \bmod m$
4.     $n \leftarrow m$
5.     $m \leftarrow r$
6. until $r = 0$
7. return $n$
```

(a) Does it matter if in the first call $gcd(m, n)$ it happens that $n < m$? Explain.
(b) Prove the correctness of Algorithm EUCLID. (Hint: Make use of the following theorem: If $r$ divides both $m$ and $n$, then $r$ divides $m - n$).
(c) Show that the running time of Algorithm EUCLID is maximum if $m$ and $n$ are two consecutive numbers in the Fibonacci sequence defined by

$$f_1 = f_2 = 1; \quad f_n = f_{n-1} + f_{n-2} \text{ for } n > 2.$$

(d) Analyze the running time of Algorithm EUCLID *in terms of* $n$, assuming that $n \geq m$.
(e) Can the time complexity of Algorithm EUCLID be expressed using the $\Theta$-notation? Explain.

**1.41.** Find the time complexity of Algorithm EUCLID discussed in Exercise 1.40 measured *in terms of the input size*. Is it logarithmic, linear, exponential? Explain.

**1.42.** Prove that for any constant $c > 0$, $(\log n)^c = o(n)$.

**1.43.** Show that any exponential function grows faster than any polynomial function by proving that for any constants $c$ and $d$ greater than 1,

$$n^c = o(d^n).$$

## 1.16 Bibliographic notes

There are several books on the design and analysis of algorithms. These include, in alphabetical order, Aho, Hopcroft, and Ullman (1974), Baase (1987), Brassard and Bratley (1988), Brassard and Bratley (1996), Dromey (1982), Horowitz and Sahni (1978), Hu (1982), Knuth (1968, 1969, 1973), Manber (1989), Mehlhorn (1984), Moret and Shapiro (1991), Purdom and Brown (1985), Reingold, Nievergelt, and Deo (1977), Sedgewick (1983) and Wilf (1986). For a more popular account of algorithms, see Knuth (1977), Lewis and Papadimitriou (1978) and the two Turing Award Lectures of Karp (1986) and Tarjan (1987). Some of the more practical aspects of algorithm design are discussed in Bentley (1982) and Gonnet (1984). Knuth (1973) discusses in detail the sorting algorithms covered in this chapter. He gives step-counting analyses. The asymptotic notation was used in mathematics before the emergence of the field of algorithms. Knuth (1976) gives an account of its history. This article discusses the $\Omega$ and $\Theta$ notations and their proper usage and is an attempt to standardize these notations. Purdom and Brown (1985) presents a comprehensive treatment of advanced techniques for analyzing algorithms with numerous examples. The main mathematical aspects of the analysis of algorithms can be found in Greene and Knuth (1981). Weide (1977) provides a survey of both elementary and advanced analysis techniques. Hofri (1987) discusses the average-case analysis of algorithms in detail.

## 2.9 Exercises

**2.1.** Let $A$ and $B$ be two sets. Prove the following properties, which are known as *De Morgan's laws*.

(a) $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

(b) $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

**2.2.** Let $A, B$ and $C$ be finite sets.

(a) Prove the *principle of inclusion-exclusion for two sets*:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

(b) Prove the *principle of inclusion-exclusion for three sets*:

$$|A \cup B \cup C| = |A|+|B|+|C|-|A \cap B|-|A \cap C|-|B \cap C|+|A \cap B \cap C|.$$

**2.3.** Show that if a relation $R$ on a set $A$ is transitive and irreflexive, then $R$ is asymmetric.

**2.4.** Let $R$ be a relation on a set $A$. Then, $R^2$ is defined as $\{(a, b) \mid (a, c) \in R \text{ and } (c, b) \in R \text{ for some } c \in A\}$. Show that if $R$ is symmetric, then $R^2$ is also symmetric.

**2.5.** Let $R$ be a *nonempty* relation on a set $A$. Show that if $R$ is symmetric and transitive, then $R$ is *not* irreflexive.

**2.6.** Let $A$ be a finite set and $P(A)$ the power set of $A$. Define the relation $R$ on the set $P(A)$ by $(X, Y) \in R$ if and only if $X \subseteq Y$. Show that $R$ is a partial order.

**2.7.** Let $A = \{1, 2, 3, 4, 5\}$ and $B = A \times A$. Define the relation $R$ on the set $B$ by $\{((x, y), (w, z)) \in B\}$ if and only if $xz = yw$.

(a) Show that $R$ is an equivalence relation.

(b) Find the equivalence classes induced by $R$.

**2.8.** Given the sets $A$ and $B$ and the function $f$ from $A$ to $B$, determine whether $f$ is one to one, onto $B$ or both (i.e. a bijection).

(a) $A = \{1, 2, 3, 4, 5\}$, $B = \{1, 2, 3, 4\}$ and $f = \{(1, 2), (2, 3), (3, 4), (4, 1), (5, 2)\}$.

(b) $A$ is the set of integers, $B$ is the set of even integers and $f(n) = 2n$.

(c) $A = B$ is the set of integers, and $f(n) = n^2$.

(d) $A = B$ is the set of real numbers with 0 excluded and $f(x) = 1/x$.

(e) $A = B$ is the set of real numbers and $f(x) = |\, x \,|$.

**2.9.** A real number $r$ is called *rational* if $r = p/q$, for some integers $p$ and $q$, otherwise it is called *irrational*. The numbers $0.25, 1.3333333\ldots$ are rational, while $\pi$ and $\sqrt{p}$, for any prime number $p$, are irrational. Use the proof by contradiction method to prove that $\sqrt{7}$ is irrational.

**2.10.** Prove that for any positive integer $n$

$$\lfloor \log n \rfloor + 1 = \lceil \log(n+1) \rceil.$$

**2.11.** Give a counterexample to disprove the assertion given in Example 2.9.

**2.12.** Use mathematical induction to show that $n! > 2^n$ for $n \geq 4$.

**2.13.** Use mathematical induction to show that a tree with $n$ vertices has exactly $n-1$ edges.

**2.14.** Prove that $\phi^n = \phi^{n-1} + \phi^{n-2}$ for all $n \geq 2$, where $\phi$ is the golden ratio (see Example 2.11).

**2.15.** Prove that for every positive integer $k$, $\sum_{i=1}^{n} i^k \log i = O(n^{k+1} \log n)$.

**2.16.** Show that

$$\sum_{j=1}^{n} j \log j = \Theta(n^2 \log n)$$

(a) using algebraic manipulations.
(b) using the method of approximating summations by integration.

**2.17.** Show that

$$\sum_{j=1}^{n} \log(n/j) = O(n),$$

(a) using algebraic manipulations.
(b) using the method of approximating summations by integration.

**2.18.** Solve the following recurrence relations.
(a) $f(n) = 3f(n-1)$ for $n \geq 1$; $f(0) = 5$.
(b) $f(n) = 2f(n-1)$ for $n \geq 1$; $f(0) = 2$.
(c) $f(n) = 5f(n-1)$ for $n \geq 1$; $f(0) = 1$.

**2.19.** Solve the following recurrence relations.
(a) $f(n) = 5f(n-1) - 6f(n-2)$ for $n \geq 2$; $f(0) = 1, f(1) = 0$.
(b) $f(n) = 4f(n-1) - 4f(n-2)$ for $n \geq 2$; $f(0) = 6, f(1) = 8$.
(c) $f(n) = 6f(n-1) - 8f(n-2)$ for $n \geq 2$; $f(0) = 1, f(1) = 0$.
(d) $f(n) = -6f(n-1) - 9f(n-2)$ for $n \geq 2$; $f(0) = 3, f(1) = -3$.
(e) $2f(n) = 7f(n-1) - 3f(n-2)$ for $n \geq 2$; $f(0) = 1, f(1) = 1$.
(f) $f(n) = f(n-2)$ for $n \geq 2$; $f(0) = 5, f(1) = -1$.

**2.20.** Solve the following recurrence relations.

(a) $f(n) = f(n-1) + n^2 \quad \text{for } n \geq 1; \quad f(0) = 0.$
(b) $f(n) = 2f(n-1) + n \quad \text{for } n \geq 1; \quad f(0) = 1.$
(c) $f(n) = 3f(n-1) + 2^n \quad \text{for } n \geq 1; \quad f(0) = 3.$
(d) $f(n) = 2f(n-1) + n^2 \quad \text{for } n \geq 1; \quad f(0) = 1.$
(e) $f(n) = 2f(n-1) + n + 4 \quad \text{for } n \geq 1; \quad f(0) = 4.$
(f) $f(n) = -2f(n-1) + 2^n - n^2 \quad \text{for } n \geq 1; \quad f(0) = 1.$
(g) $f(n) = nf(n-1) + 1 \quad \text{for } n \geq 1; \quad f(0) = 1.$

**2.21.** Consider the following recurrence

$$f(n) = 4f(n/2) + n \quad \text{for } n \geq 2; \quad f(1) = 1,$$

where $n$ is assumed to be a power of 2.
(a) Solve the recurrence by expansion.
(b) Solve the recurrence directly by applying Theorem 2.5.

**2.22.** Consider the following recurrence

$$f(n) = 5f(n/3) + n \quad \text{for } n \geq 2; \quad f(1) = 1,$$

where $n$ is assumed to be a power of 3.
(a) Solve the recurrence by expansion.
(b) Solve the recurrence directly by applying Theorem 2.5.

**2.23.** Consider the following recurrence

$$f(n) = 9f(n/3) + n^2 \quad \text{for } n \geq 2; \quad f(1) = 1,$$

where $n$ is assumed to be a power of 3.
(a) Solve the recurrence by expansion.
(b) Solve the recurrence directly by applying Theorem 2.5.

**2.24.** Consider the following recurrence

$$f(n) = 2f(n/4) + \sqrt{n} \quad \text{for } n \geq 4; \quad f(n) = 1 \text{ if } n < 4,$$

where $n$ is assumed to be of the form $2^{2^k}, k \geq 0$.
(a) Solve the recurrence by expansion.
(b) Solve the recurrence directly by applying Theorem 2.5.

**2.25.** Use the substitution method to find an upper bound for the recurrence

$$f(n) = f(\lfloor n/2 \rfloor) + f(\lfloor 3n/4 \rfloor) \quad \text{for } n \geq 4; \quad f(n) = 4 \text{ if } n < 4.$$

Express the solution using the $O$-notation.

**2.26.** Use the substitution method to find an upper bound for the recurrence

$$f(n) = f(\lfloor n/4 \rfloor) + f(\lfloor 3n/4 \rfloor) + n \quad \text{for } n \geq 4; \quad f(n) = 4 \text{ if } n < 4.$$

Express the solution using the $O$-notation.

**2.27.** Use the substitution method to find a lower bound for the recurrence in Exercise 2.25. Express the solution using the $\Omega$-notation.

**2.28.** Use the substitution method to find a lower bound for the recurrence in Exercise 2.26. Express the solution using the $\Omega$-notation.

**2.29.** Use the substitution method to solve the recurrence

$$f(n) = 2f(n/2) + n^2 \quad \text{for } n \geq 2; \quad f(1) = 1,$$

where $n$ is assumed to be a power of 2. Express the solution using the $\Theta$-notation.

**2.30.** Let

$$f(n) = f(n/2) + n \quad \text{for } n \geq 2; \quad f(1) = 1,$$

and

$$g(n) = 2g(n/2) + 1 \quad \text{for } n \geq 2; \quad g(1) = 1,$$

where $n$ is a power of 2. Is $f(n) = g(n)$? Prove your answer.

**2.31.** Use the change of variable method to solve the recurrence

$$f(n) = f(n/2) + \sqrt{n} \quad \text{for } n \geq 4; \quad f(n) = 2 \text{ if } n < 4,$$

where $n$ is assumed to be of the form $2^{2^k}$. Find the asymptotic behavior of the function $f(n)$

**2.32.** Use the change of variable method to solve the recurrence

$$f(n) = 2f(\sqrt{n}) + n \quad \text{for } n \geq 4; \quad f(n) = 1 \text{ if } n < 4,$$

where $n$ is assumed to be of the form $2^{2^k}$. Find the asymptotic behavior of the function $f(n)$

**2.33.** Prove that the solution to the recurrence

$$f(n) = 2f(n/2) + g(n) \quad \text{for } n \geq 2; \quad f(1) = 1$$

is $f(n) = O(n)$ whenever $g(n) = o(n)$. For example, $f(n) = O(n)$ if $g(n) = n^{1-\epsilon}$, $0 < \epsilon < 1$.

nodes in group $g$ is at most

$$\frac{n}{F(g)}\,(F(g) - F(g-1)) \leq n.$$

Since there are at most $\log^* n$ groups $(0, 1, \ldots, \log^* n - 1)$, it follows that the number of node charges assigned to all nodes is $O(n \log^* n)$. Combining this with the $O(m \log^* n)$ charges to the *find* instructions yields the following theorem:

**Theorem 4.3** Let $T(m)$ denote the running time required to process an interspersed sequence $\sigma$ of $m$ *union* and *find* operations using union by rank and path compression. Then $T(m) = O(m \log^* n)$ in the worst case.

Note that for almost all practical purposes, $\log^* n \leq 5$. This means that the running time is $O(m)$ for virtually all practical applications.

## 4.4 Exercises

**4.1.** What are the merits and demerits of implementing a priority queue using an ordered list?

**4.2.** What are the costs of *insert* and *delete-max* operations of a priority queue that is implemented as a regular queue.

**4.3.** Which of the following arrays are heaps?

(a) $\begin{array}{|c|c|c|c|c|}\hline 8&6&4&3&2\\\hline\end{array}$. (b) $\begin{array}{|c|}\hline 7\\\hline\end{array}$. (c) $\begin{array}{|c|c|c|c|c|}\hline 9&7&5&6&3\\\hline\end{array}$.

(d) $\begin{array}{|c|c|c|c|c|c|c|}\hline 9&4&8&3&2&5&7\\\hline\end{array}$. (e) $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 9&4&7&2&1&6&5&3\\\hline\end{array}$.

**4.4.** Where do the following element keys reside in a heap?
(a) Second largest key. (b) Third largest key. (c) Minimum key.

**4.5.** Give an efficient algorithm to test whether a given array $A[1..n]$ is a heap. What is the time complexity of your algorithm?

**4.6.** Which heap operation is more costly: insertion or deletion? Justify your answer. Recall that both operations have the same time complexity, that is, $O(\log n)$.

**4.7.** Let $H$ be the heap shown in Fig. 4.1. Show the heap that results from

(a) deleting the element with key 17.
(b) inserting an element with key 19.

**4.8.** Show the heap (in both tree and array representation) that results from deleting the maximum key in the heap shown in Fig. 4.4(e).

**4.9.** How fast is it possible to find the *minimum* key in a max-heap of $n$ elements?

**4.10.** Prove or disprove the following claim. Let $x$ and $y$ be two elements in a heap whose keys are positive integers, and let $T$ be the tree representing that heap. Let $h_x$ and $h_y$ be the heights of $x$ and $y$ in $T$. Then, if $x$ is greater than $y$, $h_x$ cannot be less than $h_y$. (See Sec. 3.5 for the definition of node height).

**4.11.** Illustrate the operation of Algorithm MAKEHEAP on the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 7 | 2 | 1 | 9 | 8 | 6 | 4 |

.

**4.12.** Show the steps of transforming the following array into a heap

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 3 | 2 | 5 | 7 | 6 | 8 |

.

**4.13.** Let $A[1..19]$ be an array of 19 integers, and suppose we apply Algorithm MAKEHEAP on this array.

(a) How many calls to Procedure SIFT-DOWN will there be? Explain.

(b) What is the maximum number of element interchanges in this case? Explain.

(c) Give an array of 19 elements that requires the above maximum number of element interchanges.

**4.14.** Show how to use Algorithm HEAPSORT to arrange in increasing order the integers in the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 2 | 9 | 8 | 7 | 1 | 3 |

.

**4.15.** Given an array $A[1..n]$ of integers, we can create a heap $B[1..n]$ from $A$ as follows. Starting from the empty heap, repeatedly insert the elements of $A$ into $B$, each time adjusting the current heap, until $B$ contains all the elements in $A$. Show that the running time of this algorithm is $\Theta(n \log n)$ in the worst case.

**4.16.** Illustrate the operation of the algorithm in Exercise 4.15 on the array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 9 | 2 | 7 | 1 | 8 | 4 | 3 |

.

**4.17.** Explain the behavior of Algorithm HEAPSORT when the input array is already sorted in

(a) increasing order.

(b) decreasing order.

**4.18.** Give an example of a binary search tree with the heap property.

**4.19.** Give an algorithm to merge two heaps of the same size into one heap. What is the time complexity of your algorithm?

**4.20.** Compute the minimum and maximum number of element comparisons performed by Algorithm HEAPSORT.

**4.21.** A $d$-heap is a generalization of the binary heap discussed in this chapter. It is represented by an almost-complete $d$-ary rooted tree for some $d \geq 2$. Rewrite Procedure SIFT-UP for the case of $d$-heaps. What is its time complexity?

**4.22.** Rewrite Procedure SIFT-DOWN for the case of $d$-heaps (see Exercise 4.21). What is its time complexity measured in terms of $d$ and $n$?

**4.23.** Give a sequence of $n$ *union* and *find* operations that results in a tree of height $\Theta(\log n)$ using only the heuristic of union by rank. Assume the set of elements is $\{1, 2, \ldots, n\}$.

**4.24.** Give a sequence of $n$ *union* and *find* operations that requires $\Theta(n \log n)$ time using only the heuristic of union by rank. Assume the set of elements is $\{1, 2, \ldots, n\}$.

**4.25.** What are the ranks of nodes 3, 4 and 8 in Fig. 4.8(f)?

**4.26.** Let $\{1\}, \{2\}, \{3\}, \ldots, \{8\}$ be $n$ singleton sets, each represented by a tree with exactly one node. Use the union-find algorithms with union by rank and path compression to find the tree representation of the set resulting from each of the following *union*s and *find*s: UNION$(1, 2)$, UNION$(3, 4)$,UNION$(5, 6)$, UNION$(7, 8)$, UNION$(1, 3)$, UNION$(5, 7)$, FIND$(1)$, UNION$(1, 5)$, FIND$(1)$.

**4.27.** Let $T$ be a tree resulting from a sequence of *union*s and *find*s using both the heuristics of union by rank and path compression, and let $x$ be a node in $T$. Prove that $rank(x)$ is an upper bound on the height of $x$.

**4.28.** Let $\sigma$ be a sequence of *union* and *find* instructions in which all the *union*s occur before the *find*s. Show that the running time is linear if both the heuristics of union by rank and path compression are used.

**4.29.** Another heuristic that is similar to union by rank is the *weight-balancing rule*. In this heuristic, the action of the operation UNION$(x, y)$ is to let the root of the tree with fewer nodes point to the root of the tree with a larger number of nodes. If both trees have the same number of nodes, then let $y$ be the parent of $x$. Compare this heuristic with the union by rank heuristic.

**4.30.** Solve Exercise 4.26 using *the weight-balancing rule and path compression* (see Exercise 4.29).

**4.31.** Prove that the weight-balancing rule described in Exercise 4.29 guarantees that the resulting tree is of height $O(\log n)$.

**4.32.** Let $T$ be a tree resulting from a sequence of *union*s and *find*s using the heuristics of union by rank and path compression. Let $x$ be the root of $T$ and $y$ a leaf node in $T$. Prove that the ranks of the nodes on the path from $y$ to $x$ form a *strictly* increasing sequence.

**4.33.** Prove the observation that if node $v$ is in rank group $g > 0$, then $v$ can be moved and charged at most $F(g) - F(g-1)$ times before it acquires a parent in a higher group.

**4.34.** Another possibility for the representation of disjoint sets is by using linked lists. Each set is represented by a linked list, where the set representative is the first element in the list. Each element in the list has a pointer to the set representative. Initially, one list is created for each element. The union of two sets is implemented by merging the two sets. Suppose two sets $S_1$ represented by list $L_1$ and $S_2$ represented by list $L_2$ are to be merged. If the first element in $L_1$ is to be used as the name of the resulting set, then the pointer to the set name at each element in $L_2$ must be changed so that it points to the first element in $L_1$.

(a) Explain how to improve this representation so that each *find* operation takes $O(1)$ time.

(b) Show that the total cost of performing $n-1$ *union*s is $\Theta(n^2)$ in the worst case.

**4.35.** (Refer to Exercise 4.34). Show that if when performing the union of two sets, the first element in the list with a larger number of elements is always chosen as the name of the new set, then the total cost of performing $n-1$ *union*s becomes $O(n \log n)$.

## 4.5 Bibliographic notes

Heaps and the data structures for disjoint sets appear in several books on algorithms and data structures (see the bibliographic notes of chapters 1 and 3). They are covered in greater depth in Tarjan (1983). Heaps were first introduced as part of heapsort by Williams (1964). The linear time algorithm for building a heap is due to Floyd (1964). A number of variants of heaps can be found in Cormen *et al.* (1992), e.g. binomial heaps, Fibonacci heaps. A comparative study of many data structures for priority queues can be found in Jones (1986). The disjoint sets data structure was first studied by Galler and Fischer (1964) and Fischer (1972). A more detailed analysis was carried out by Hopcroft and Ullman (1973) and then a more exact analysis by Tarjan (1975). In this paper, a lower bound that is *not* linear was established when both union by rank and path compression are used.

the majority must be the median, we can scan the sequence to test if the median is indeed the majority. This method takes $\Theta(n)$ time, as the median can be found in $\Theta(n)$ time. As we will see in Sec. 6.5, the hidden constant in the time complexity of the median finding algorithm is too large, and the algorithm is fairly complex.

It turns out that there is an elegant solution that uses much fewer comparisons. We derive this algorithm using induction. The essence of the algorithm is based on the following observation:

**Observation 5.1** If two *different* elements in the original sequence are removed, then the majority in the original sequence remains the majority in the new sequence.

This observation suggests the following procedure for finding an element that is a *candidate* for being the majority. Set a counter to zero and let $x = A[1]$. Starting from $A[2]$, scan the elements one by one increasing the counter by one if the current element is equal to $x$ and decreasing the counter by one if the current element is not equal to $x$. If all the elements have been scanned and the counter is greater than zero, then return $x$ as the candidate. If the counter becomes zero when comparing $x$ with $A[j]$, $1 < j < n$, then call procedure *candidate* recursively on the elements $A[j+1..n]$. Notice that decrementing the counter implements the idea of throwing two different elements as stated in Observation 5.1. This method is described more precisely in Algorithm MAJORITY. Converting this recursive algorithm into an iterative one is straightforward, and is left as an exercise.

## 5.8 Exercises

**5.1.** Give a recursive algorithm that computes the $n$th Fibonacci number $f_n$ defined by

$$f_1 = f_2 = 1; \quad f_n = f_{n-1} + f_{n-2} \text{ for } n \geq 3.$$

**5.2.** Give an iterative algorithm that computes the $n$th Fibonacci number $f_n$ defined above.

**5.3.** Use induction to develop a recursive algorithm for finding the maximum element in a given sequence $A[1..n]$ of $n$ elements.

**Algorithm 5.9** MAJORITY
**Input:** An array $A[1..n]$ of $n$ elements.
**Output:** The majority element if it exists; otherwise *none*.

```
1. $c \leftarrow candidate(1)$
2. $count \leftarrow 0$
3. for $j \leftarrow 1$ to $n$
4.     if $A[j] = c$ then $count \leftarrow count + 1$
5. end for
6. if $count > \lfloor n/2 \rfloor$ then return $c$
7. else return $none$
```

**Procedure** *candidate*$(m)$

```
1. $j \leftarrow m$;   $c \leftarrow A[m]$;   $count \leftarrow 1$
2. while $j < n$ and $count > 0$
3.     $j \leftarrow j + 1$
4.     if $A[j] = c$ then $count \leftarrow count + 1$
5.     else $count \leftarrow count - 1$
6. end while
7. if $j = n$ then return $c$      {See Exercises 5.31 and 5.32.}
8. else return $candidate(j + 1)$
```

**5.4.** Use induction to develop a recursive algorithm for finding the average of $n$ real numbers $A[1..n]$.

**5.5.** Use induction to develop a recursive algorithm that searches for an element $x$ in a given sequence $A[1..n]$ of $n$ elements.

**5.6.** Derive the running time of Algorithm INSERTIONSORTREC.

**5.7.** Illustrate the operation of Algorithm RADIXSORT on the following sequence of eight numbers:

(a) 4567, 2463, 6523, 7461, 4251, 3241, 6492, 7563.
(b) 16543, 25895, 18674, 98256, 91428, 73234, 16597, 73195.

**5.8.** Express the time complexity of Algorithm RADIXSORT in terms of $n$ when the input consists of $n$ positive integers in the interval

(a) $[1..n]$.
(b) $[1..n^2]$.
(c) $[1..2^n]$.

**5.9.** Let $A[1..n]$ be an array of positive integers in the interval $[1..n!]$. Which sorting algorithm do you think is faster: BOTTOMUPSORT or RADIXSORT? (See Sec. 1.7).

**5.10.** What is the time complexity of Algorithm RADIXSORT if arrays are used instead of linked lists? Explain.

**5.11.** Give a recursive version of Algorithm BUBBLESORT given in Exercise 1.16.

**5.12.** A sorting method known as *bucket sort* works as follows. Let $A[1..n]$ be a sequence of $n$ numbers within a reasonable range, say all numbers are between 1 and $m$, where $m$ is not too large compared to $n$. The numbers are distributed into $k$ buckets, with the first bucket containing those numbers between 1 and $\lfloor m/k \rfloor$, the second bucket containing those numbers between $\lfloor m/k \rfloor + 1$ to $\lfloor 2m/k \rfloor$, and so on. The numbers in each bucket are then sorted using another sorting algorithm, say Algorithm INSERTIONSORT. Analyze the running time of the algorithm.

**5.13.** Instead of using another sorting algorithm in Exercies 5.12, design a recursive version of bucket sort that recursively sorts the numbers in each bucket. What is the major disadvantage of this recursive version?

**5.14.** A sorting algorithm is called *stable* if the order of equal elements is preserved after sorting. Which of the following sorting algorithms are stable?

(a)SELECTIONSORT (b)INSERTIONSORT (c)BUBBLESORT
(d)BOTTOMUPSORT (e)HEAPSORT (f)RADIXSORT.

**5.15.** Use induction to solve Exercise 3.7.

**5.16.** Use induction to solve Exercise 3.8.

**5.17.** Use Horner's rule described in Sec. 5.5 to evaluate the following polynomials:

(a) $3x^5 + 2x^4 + 4x^3 + x^2 + 2x + 5$.
(b) $2x^7 + 3x^5 + 2x^3 + 5x^2 + 3x + 7$.

**5.18.** Use Algorithm EXPREC to compute
(a) $2^5$. (b) $2^7$. (c) $3^5$. (d) $5^7$.

**5.19.** Solve Exercise 5.18 using Algorithm EXP instead of Algorithm EXPREC.

**5.20.** Carefully explain why in Algorithm PERMUTATIONS1 when $P[j]$ and $P[m]$ are interchanged before the recursive call, they must be interchanged back after the recursive call.

**5.21.** Carefully explain why in Algorithm PERMUTATIONS2 $P[j]$ must be reset to 0 after the recursive call.

**5.22.** Carefully explain why in Algorithm PERMUTATIONS2, when Procedure *perm2* is invoked by the call *perm2*$(m)$ with $m > 0$, the array $P$ contains *exactly* $m$ zeros, and hence the recursive call *perm2*$(m-1)$ will be executed *exactly* $m$ times.

**5.23.** Modify Algorithm PERMUTATIONS2 so that the permutations of the numbers $1, 2, \ldots, n$ are generated in a reverse order to that produced by Algorithm PERMUTATIONS2.

**5.24.** Modify Algorithm PERMUTATIONS2 so that it generates all $k$-subsets of the set $\{1, 2, \ldots, n\}, 1 \leq k \leq n$.

**5.25.** Analyze the time complexity of the modified algorithm in Exercise 5.24.

**5.26.** Prove the correctness of Algorithm PERMUTATIONS1.

**5.27.** Prove the correctness of Algorithm PERMUTATIONS2.

**5.28.** Give an iterative version of Algorithm MAJORITY.

**5.29.** Illustrate the operation of Algorithm MAJORITY on the arrays

(a) $\boxed{5 \mid 7 \mid 5 \mid 4 \mid 5}$.

(b) $\boxed{5 \mid 7 \mid 5 \mid 4 \mid 8}$.

(c) $\boxed{2 \mid 4 \mid 1 \mid 4 \mid 4 \mid 4 \mid 6 \mid 4}$.

**5.30.** Prove Observation 5.1.

**5.31.** Prove or disprove the following claim. If in Step 7 of Procedure *candidate* in Algorithm MAJORITY $j = n$ but $count = 0$ then $c$ is the majority element.

**5.32.** Prove or disprove the following claim. If in Step 7 of Procedure *candidate* in Algorithm MAJORITY $j = n$ and $count > 0$ then $c$ is the majority element.

**5.33.** Let $A[1..n]$ be a *sorted* array of $n$ integers, and $x$ an integer. Design an $O(n)$ time algorithm to determine whether there are two elements in $A$, if any, whose sum is exactly $x$.

## 5.9 Bibliographic notes

The use of induction as a mathematical technique for proving the correctness of algorithms was first developed by Floyd (1967). Recursion has been studied extensively in algorithm design. See for example the books of Burge (1975) and Paull (1988). The use of induction as a design technique appears in Manber (1988). Manber (1989) is a whole book that is mostly devoted to the induction design technique. Unlike this chapter, induction in that book encompasses a wide variety of problems and is used in its broad sense to cover other design techniques like divide and conquer and dynamic programming. Radix sort is used by card-sorting machines. In old machines, the machine did the distribution step and the operator collected the piles after each pass and combined them into one for the next pass. Horner's rule for polynomial evaluation is after the English mathematician W. G. Horner. Algorithm PERMUTATIONS2 appears in Banachowski, Kreczmar duce the time taken by the combine step to $\Theta(n)$, then the time complexity of the algorithm will be $\Theta(n \log n)$. This can be achieved by a process called *presorting*, i.e., the elements in $S$ are sorted by their $y$-coordinates once and for all and stored in an array $Y$. Each time we need to sort $T$ in the combine step, we only need to extract its elements from $Y$ in $\Theta(n)$ time. This is easy to do since the points in $T$ are those points in $Y$ within distance $\delta$ from the vertical line $L$. This modification reduces the time required by the combine step to $\Theta(n)$. Thus, the recurrence relation becomes

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ 3 & \text{if } n = 3 \\ 2T(n/2) + \Theta(n) & \text{if } n > 3. \end{cases}$$

The solution to this familiar recurrence is the desired $\Theta(n \log n)$ bound. The above discussion implies Algorithm CLOSESTPAIR. In the algorithm, for a point $p$, $x(p)$ denotes the $x$-coordinate of point $p$.

The following theorem summarizes the main result. Its proof is embedded in the description of the algorithm and the analysis of its running time.

**Theorem 6.7** Given a set $S$ of $n$ points in the plane, Algorithm CLOSESTPAIR finds a pair of points in $S$ with minimum separation in $\Theta(n \log n)$ time.

## 6.10 Exercises

**6.1.** Modify Algorithm MINMAX so that it works when $n$ is not a power of 2. Is the number of comparisons performed by the new algorithm $\lfloor 3n/2 - 2 \rfloor$ even if $n$ is not a power of 2? Prove your answer.

**6.2.** Consider Algorithm SLOWMINMAX which is obtained from Algorithm MINMAX by replacing the test

**if** $high - low = 1$

by the test

**if** $high = low$

and making some other changes in the algorithm accordingly. Thus, in Algorithm SLOWMINMAX, the recursion is halted when the size of the input array is 1. Count the number of comparisons required by this algorithm to find the minimum and maximum of an array $A[1..n]$, where

**Algorithm 6.7** CLOSESTPAIR
**Input:** A set $S$ of $n$ points in the plane.

**Output:** The minimum separation realized by two points in $S$.

```
1. Sort The points in $S$ in nondecreasing order of their $x$-coordinates.
2. $Y \leftarrow$ The points in $S$ sorted in nondecreasing order of their $y$-
   coordinates.
3. $\delta \leftarrow cp(1, n)$

Procedure $cp(low, high)$
 1. if $high - low + 1 \leq 3$ then compute $\delta$ by a straightforward method.
 2. else
 3.     $mid \leftarrow \lfloor (low + high)/2 \rfloor$
 4.     $x_0 \leftarrow x(S[mid])$
 5.     $\delta_l \leftarrow cp(low, mid)$
 6.     $\delta_r \leftarrow cp(mid + 1, high)$
 7.     $\delta \leftarrow \min\{\delta_l, \delta_r\}$
 8.     $k \leftarrow 0$
 9.     for $i \leftarrow 1$ to $n$      {Extract $T$ from $Y$}
10.         if $|x(Y[i]) - x_0| \leq \delta$ then
11.             $k \leftarrow k + 1$
12.             $T[k] \leftarrow Y[i]$
13.         end if
14.     end for      {$k$ is the size of $T$}
15.     $\delta' \leftarrow 2\delta$          {Initialize $\delta'$ to any number greater than $\delta$}
16.     for $i \leftarrow 1$ to $k - 1$      {Compute $\delta'$}
17.         for $j \leftarrow i + 1$ to $\min\{i + 7, k\}$
18.             if $d(T[i], T[j]) < \delta'$ then $\delta' \leftarrow d(T[i], T[j])$
19.         end for
20.     end for
21.     $\delta \leftarrow \min\{\delta, \delta'\}$
22. end if
23. return $\delta$
```

$n$ is a power of 2. Explain why the number of comparisons in this algorithm is greater than that in Algorithm MINMAX. (Hint: In this case, the initial condition is $C(1) = 0$).

**6.3.** Derive an iterative minimax algorithm that finds both the minimum and maximum in a set of $n$ elements using only $3n/2 - 2$ comparisons, where $n$ is a power of 2.

**6.4.** Give a divide-and-conquer version of Algorithm LINEARSEARCH given in Sec. 1.3. The algorithm should start by dividing the input elements into approximately two halves. How much work space is required by the algorithm?

**6.5.** Give a divide-and-conquer algorithm to find the sum of all numbers in an array $A[1..n]$ of integers. The algorithm should start by dividing the input elements into approximately two halves. How much work space is required by the algorithm?

**6.6.** Let $A[1..n]$ be an array of $n$ integers and $x$ an integer. Derive a divide-and-conquer algorithm to find the frequency of $x$ in $A$, i.e., the number of times $x$ appears in $A$. What is the time complexity of your algorithm?

**6.7.** Modify Algorithm BINARYSEARCHREC so that it searches for two keys. In other words, given an array $A[1..n]$ of $n$ elements and two elements $x_1$ and $x_2$, the algorithm should return two integers $k_1$ and $k_2$ representing the positions of $x_1$ and $x_2$, respectively, in $A$.

**6.8.** Design a search algorithm that divides a sorted array into one third and two thirds instead of two halves as in Algorithm BINARYSEARCHREC. Analyze the time complexity of the algorithm.

**6.9.** Modify Algorithm BINARYSEARCHREC so that it divides the sorted array into three equal parts instead of two as in Algorithm BINARYSEARCHREC. In each iteration, the algorithm should test the element $x$ to be searched for against two entries in the array. Analyze the time complexity of the algorithm.

**6.10.** Use Algorithm MERGESORT to sort the array

(a) $\boxed{32}\boxed{15}\boxed{14}\boxed{15}\boxed{11}\boxed{17}\boxed{25}\boxed{51}$ .

(b) $\boxed{12}\boxed{25}\boxed{17}\boxed{19}\boxed{51}\boxed{32}\boxed{45}\boxed{18}\boxed{22}\boxed{37}\boxed{15}$ .

**6.11.** Use mathematical induction to prove the correctness of Algorithm MERGESORT. Assume that Algorithm MERGE works correctly.

**6.12.** Show that the space complexity of Algorithm MERGESORT is $\Theta(n)$.

**6.13.** It was shown in Sec. 6.3 that algorithms BOTTOMUPSORT and MERGESORT are very similar. Give an example of an array of numbers in which

(a) Algorithm BOTTOMUPSORT and Algorithm MERGESORT perform the same number of element comparisons.

(b) Algorithm BOTTOMUPSORT performs more element comparisons than Algorithm MERGESORT.

(c) Algorithm BOTTOMUPSORT performs fewer element comparisons than Algorithm MERGESORT.

**6.14.** Consider the following modification of Algorithm MERGESORT. The algorithm first divides the input array $A[low..high]$ into four parts $A_1, A_2, A_3$ and $A_4$ instead of two. It then sorts each part recursively, and finally merges the four sorted parts to obtain the original array in sorted order. Assume for simplicity that $n$ is a power of 4.

(a) Write out the modified algorithm.

(b) Analyze its running time.

**6.15.** What will be the running time of the modified algorithm in Exercise 6.14 if the input array is divided into $k$ parts instead of 4? Here, $k$ is a *fixed* positive integer greater than 1.

**6.16.** Consider the following modification to Algorithm MERGESORT. We apply the algorithm on the input array $A[1..n]$ and continue the recursive calls until the size of a subinstance becomes relatively small, say $m$ or less. At this point, we switch to Algorithm INSERTIONSORT and apply it on the small instance. So, the first test of the modified algorithm will look like the following:

**if** $\mathit{high} - \mathit{low} + 1 \leq m$ **then** INSERTIONSORT($A[\mathit{low}..\mathit{high}]$).

What is the largest value of $m$ in terms of $n$ such that the running time of the modified algorithm will still be $\Theta(n \log n)$? You may assume for simplicity that $n$ is a power of 2.

**6.17.** Use Algorithm SELECT to find the $k$th smallest element in the list of numbers given in Example 6.1, where

(a) $k = 1$. (b) $k = 9$. (c) $k = 17$. (d) $k = 22$. (e) $k = 25$.

**6.18.** What will happen if in Algorithm SELECT the true median of the elements is chosen as the pivot instead of the median of medians? Explain.

**6.19.** Let $A[1..105]$ be a *sorted* array of 105 integers. Suppose we run Algorithm SELECT to find the 17th element in $A$. How many recursive calls to Procedure *select* will there be? Explain your answer clearly.

**6.20.** Explain the behavior of Algorithm SELECT if the input array is already sorted in nondecreasing order. Compare that to the behavior of Algorithm BINARYSEARCHREC.

**6.21.** In Algorithm SELECT, groups of size 5 are sorted in each invocation of the algorithm. This means that finding a procedure that sorts a group of size 5 that uses the fewest number of comparisons is important. Show that it is possible to sort five elements using only seven comparisons.

**6.22.** One reason that Algorithm SELECT is inefficient is that it does not make full use of the comparisons that it makes: After it discards one portion of the elements, it starts on the subproblem from scratch. Give a precise count of the number of comparisons the algorithm performs when presented with $n$ elements. Note that it is possible to sort five elements using only seven comparisons (see Exercise 6.21).

**6.23.** Based on the number of comparisons counted in Exercise 6.22, determine for what values of $n$ one should use a straightforward sorting method and extract the $k$th element directly.

**6.24.** Let $g$ denote the size of each group in Algorithm SELECT for some positive integer $g \geq 3$. Derive the running time of the algorithm in terms of $g$.

What happens when $g$ is too large compared to the value used in the algorithm, namely 5?

**6.25.** Which of the following group sizes 3, 4, 5, 7, 9, 11 guarantees $\Theta(n)$ worst case performance for Algorithm SELECT? Prove your answer. (See Exercise 6.24).

**6.26.** Rewrite Algorithm SELECT using Algorithm SPLIT to partition the input array. Assume for simplicity that all input elements are distinct. What is the advantage of the modified algorithm?

**6.27.** Let $A[1..n]$ and $B[1..n]$ be two arrays of distinct integers sorted in increasing order. Give an efficient algorithm to find the median of the $2n$ elements in both $A$ and $B$. What is the running time of your algorithm?

**6.28.** Make use of the algorithm obtained in Exercise 6.27 to device a divide-and-conquer algorithm for finding the median in an array $A[1..n]$. What is the time complexity of your algorithm? (Hint: Make use of Algorithm MERGESORT).

**6.29.** Consider the problem of finding *all* the first $k$ smallest elements in an array $A[1..n]$ of $n$ *distinct* elements. Here, $k$ is *not* constant, i.e., it is part of the input. We can solve this problem easily by sorting the elements and returning $A[1..k]$. This, however, costs $O(n \log n)$ time. Give a $\Theta(n)$ time algorithm for this problem. Note that running Algorithm SELECT $k$ times costs $\Theta(kn) = O(n^2)$ time, as $k$ is not constant.

**6.30.** Consider the *multiselection* problem: Given a set $S$ of $n$ elements and a set $K$ of $r$ ranks $k_1, k_2, \ldots, k_r$, find the $k_1$th, $k_2$th, ..., $k_r$th smallest elements. For example, if $K = \{2, 7, 9, 50\}$, the problem is to find the 2nd, 7th, 9th and 50th smallest elements. This problem can be solved trivially in $\Theta(rn)$ time by running Algorithm SELECT $r$ times, once for each rank $k_j, 1 \leq j \leq r$. Give an $O(n \log r)$ time algorithm to solve this problem.

**6.31.** Apply Algorithm SPLIT on the array $\boxed{27}\boxed{13}\boxed{31}\boxed{18}\boxed{45}\boxed{16}\boxed{17}\boxed{53}$ .

**6.32.** Let $f(n)$ be the number of element interchanges that Algorithm SPLIT makes when presented with the input array $A[1..n]$ excluding interchanging $A[low]$ with $A[i]$.

(a) For what input arrays $A[1..n]$ is $f(n) = 0$?
(b) What is the maximum value of $f(n)$? Explain when this maximum is achieved?

**6.33.** Modify Algorithm SPLIT so that it partitions the elements in $A[low..high]$ around $x$, where $x$ is the median of $\{A[low], A[\lfloor(low + high)/2\rfloor], A[high]\}$. Will this improve the running time of Algorithm QUICKSORT? Explain.

**6.34.** Algorithm SPLIT is used to partition an array $A[low..high]$ around $A[low]$.

Another algorithm to achieve the same result works as follows. The algorithm has two pointers $i$ and $j$. Initially, $i = low$ and $j = high$. Let the pivot be $x = A[low]$. The pointers $i$ and $j$ move from left to right and from right to left, respectively, until it is found that $A[i] > x$ and $A[j] \leq x$. At this point $A[i]$ and $A[j]$ are interchanged. This process continues until $i \geq j$. Write out the complete algorithm. What is the number of comparisons performed by the algorithm?

**6.35.** Let $A[1..n]$ be a set of integers. Give an algorithm to reorder the elements in $A$ so that all negative integers are positioned to the left of all nonnegative integers. Your algorithm should run in time $\Theta(n)$.

**6.36.** Use Algorithm QUICKSORT to sort the array

(a)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 24 | 33 | 24 | 45 | 12 | 12 | 24 | 12 |

.

(b)

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 7 |

.

(c)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 32 | 27 | 18 | 45 | 11 | 63 | 12 | 19 | 16 | 25 | 52 | 14 |

.

**6.37.** Show that the work space needed by Algorithm QUICKSORT varies between $\Theta(\log n)$ and $\Theta(n)$. What is its average space complexity?

**6.38.** Explain the behavior of Algorithm QUICKSORT when the input is already sorted in decreasing order. You may assume that the input elements are all distinct.

**6.39.** Explain the behavior of Algorithm QUICKSORT when the input array $A[1..n]$ consists of $n$ identical elements.

**6.40.** Modify Algorithm QUICKSORT slightly so that it will solve the selection problem. What is the time complexity of the new algorithm in the worst case and on the average?

**6.41.** Give an iterative version of Algorithm QUICKSORT.

**6.42.** Which of the following sorting algorithms are stable (see Exercise 5.14)?

(a)HEAPSORT (b)MERGESORT (c)QUICKSORT.

**6.43.** A sorting algorithm is called *adaptive* if its running time depends not only on the number of elements $n$, but also on their order. Which of the following sorting algorithms are adaptive?

(a)SELECTIONSORT (b)INSERTIONSORT (c)BUBBLESORT (d)HEAPSORT
(e)BOTTOMUPSORT (f)MERGESORT (g)QUICKSORT (h)RADIXSORT.

**6.44.** Let $x = a + bi$ and $y = c + di$ be two complex numbers. The product $xy$ can easily be calculated using four multiplications, that is, $xy = (ac - bd) + (ad + bc)i$. Devise a method for computing the product $xy$ using only three multiplications.

**6.45.** Write out an algorithm for the traditional algorithm for matrix multiplication described in Sec. 6.8.

**6.46.** Show that the traditional algorithm for matrix multiplication described in Sec. 6.8 requires $n^3$ multiplications and $n^3 - n^2$ additions (see Exercise 6.45).

**6.47.** Explain how to modify Strassen's algorithm for matrix multiplication so that it can also be used with matrices whose size is not necessarily a power of 2.

**6.48.** Suppose we modify the algorithm for the closest pair problem so that not each point in $T$ is compared with seven points in $T$. Instead, every point to the left of the vertical line $L$ is compared with a number of points to its right.

(a) What are the necessary modifications to the algorithm?
(b) How many points to the right of $L$ have to be compared with every point to its left? Explain.

**6.49.** Rewrite the algorithm for the closest pair problem without the presorting step. The time complexity of your algorithm should be $\Theta(n \log n)$. (Hint: Make use of Algorithm MERGESORT).

**6.50.** Design a divide-and-conquer algorithm to determine whether two given binary trees $T_1$ and $T_2$ are identical.

**6.51.** Design a divide-and-conquer algorithm that computes the height of a binary tree.

**6.52.** Give a divide-and-conquer algorithm to find the second largest element in an array of $n$ numbers. Derive the time complexity of your algorithm.

**6.53.** Consider the following algorithm that attempts to find a minimum cost spanning tree $MST(G)$ for a weighted undirected graph $G = (V, E)$ (see Sec. 8.3). Divide $G$ into two subgraphs $G_1$ and $G_2$ of approximately the same number of vertices. Compute $T_1 = MST(G_1)$ and $T_2 = MST(G_2)$. Find an edge $e$ of minimum weight that connects $G_1$ with $G_2$. Return $T_1 \cup T_2 \cup \{e\}$. Show that this algorithm does not always compute a spanning tree of minimum weight. What is the shape of the spanning tree computed by the algorithm?

**6.54.** Let $B$ be an $n \times n$ chessboard, where $n$ is a power of 2. Use a divide-and-conquer argument to describe (in words) how to cover all squares of $B$ except one with L-shaped tiles. For example, if $n = 2$, then there are four squares three of which can be covered by one L-shaped tile, and if $n = 4$, then there are 16 squares of which 15 can be covered by 5 L-shaped tiles.

**6.55.** Use a combinatorial argument to show that if $n$ is a power of 2, then $n^2 \equiv 1 \pmod 3$. (Hint: Use the result of Exercise 6.54).

The $i$th entry of column 9, that is, $V[i, 9]$ contains the maximum value we can get by filling the knapsack using the first $i$ items. Thus, an optimal packing is found in the last entry of the last column and is achieved by packing items 3 and 4. There is also another optimal solution, which is packing items 1, 2 and 3. This packing corresponds to entry $V[3, 9]$ in the table, which is the optimal packing before the fourth item was considered.

## 7.7 Exercises

**7.1.** We have defined the dynamic programming paradigm in such a way that it encompasses all algorithms that solve a problem by breaking it down into smaller subproblems, saving the solution to each subproblem and using these solutions to compute an optimal solution to the main problem. Which of the following algorithms can be classified as dynamic programming algorithms?

(a) Algorithm LINEARSEARCH.
(b) Algorithm INSERTIONSORT.
(c) Algorithm BOTTOMUPSORT.
(d) Algorithm MERGESORT.

**7.2.** Give an efficient algorithm to compute $f(n)$, the $n$th number in the Fibonacci sequence (see Example 7.1). What is the time complexity of your algorithm? Is it an exponential algorithm? Explain.

**7.3.** Give an efficient algorithm to compute the binomial coefficient $\binom{n}{k}$ (see Example 7.2). What is the time complexity of your algorithm? Is it an exponential algorithm? Explain.

**7.4.** Prove Observation 7.1.

**7.5.** Use Algorithm LCS to find the length of a longest common subsequence of the two strings $A =$ "xzyzzyx" and $B =$ "zxyyzxz". Give one longest common subsequence.

**7.6.** Show how to modify Algorithm LCS so that it outputs a longest common subsequence as well.

**7.7.** Show how to modify Algorithm LCS so that it requires only $\Theta(\min\{m, n\})$ space.

**7.8.** In Sec. 7.3, it was shown that the number of ways to fully parenthesize

$n$ matrices is given by the summation

$$f(n) = \sum_{k=1}^{n-1} f(k)f(n-k).$$

Show that the solution to this recurrence is

$$f(n) = \frac{1}{n}\binom{2n-2}{n-1}.$$

**7.9.** Consider using Algorithm MATCHAIN to multiply the following five matrices:

$$M_1 : 4 \times 5, \quad M_2 : 5 \times 3, \quad M_3 : 3 \times 6, \quad M_4 : 6 \times 4, \quad M_5 : 4 \times 5.$$

Assume the intermediate results shown in Fig. 7.6 for obtaining the multiplication $M_1 \times M_2 \times M_3 \times M_4 \times M_5$, where $C[i, j]$ is the minimum number of scalar multiplications needed to carry out the multiplication $M_i \times \ldots \times M_j, 1 \leq i \leq j \leq 5$. Also shown in the figure parenthesized expressions showing the optimal sequence for carrying out the multiplication $M_i \times \ldots \times M_j$. Find $C[1, 5]$ and the optimal parenthesized expressions for carrying out the multiplication $M_1 \times \ldots \times M_5$.

| | | | | |
|---|---|---|---|---|
| $C[1,1] = 0$ $M_1$ | $C[1,2] = 60$ $M_1M_2$ | $C[1,3] = 132$ $(M_1M_2)M_3$ | $C[1,4] = 180$ $(M_1M_2)(M_3M_4)$ | |
| | $C[2,2] = 0$ $M_2$ | $C[2,3] = 90$ $M_2M_3$ | $C[2,4] = 132$ $M_2(M_3M_4)$ | $C[2,5] = 207$ $M_2((M_3M_4)M_5)$ |
| | | $C[3,3] = 0$ $M_3$ | $C[3,4] = 72$ $M_3M_4$ | $C[3,5] = 132$ $(M_3M_4)M_5$ |
| | | | $C[4,4] = 0$ $M_4$ | $C[4,5] = 120$ $M_4M_5$ |
| | | | | $C[5,5] = 0$ $M_5$ |

Fig. 7.6 An incomplete table for the matrix chain multiplication problem.

**7.10.** Give a parenthesized expression for the optimal order of multiplying the five matrices in Example 7.4.

**7.11.** Consider applying Algorithm MATCHAIN on the following five matrices:

$$M_1 : 2 \times 3, \quad M_2 : 3 \times 6, \quad M_3 : 6 \times 4, \quad M_4 : 4 \times 2, \quad M_5 : 2 \times 7.$$

(a) Find the minimum number of scalar multiplications needed to multiply the five matrices, (that is $C[1, 5]$).
(b) Give a parenthesized expression for the order in which this optimal number of multiplications is achieved.

**7.12.** Give an example of three matrices in which one order of their multiplication costs at least 100 times the other order.

**7.13.** Show how to modify the matrix chain multiplication algorithm so that it also produces the order of multiplications as well.

**7.14.** Let $G = (V, E)$ be a weighted directed graph, and let $s, t \in V$. Assume that there is at least one path from $s$ to $t$;

(a) Let $\pi$ be a path of shortest length from $s$ to $t$ that passes by another vertex $x$. Show that the portion of the path from $s$ to $x$ is a shortest path from $s$ to $x$.
(b) Let $\pi'$ be a longest simple path from $s$ to $t$ that passes by another vertex $y$. Show that the portion of the path from $s$ to $y$ is not necessarily a longest path from $s$ to $y$.

**7.15.** Run the all-pairs shortest path algorithm on the weighted directed graph shown in Fig. 7.7.

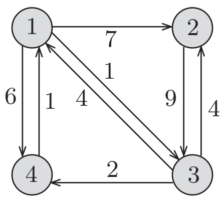

Fig. 7.7 An instance of the all-pairs shortest path problem.

**7.16.** Use the all-pairs shortest path algorithm to compute the distance matrix for the directed graph with the lengths of the edges between all pairs of vertices are as given by the matrix

$$(a) \begin{bmatrix} 0 & 1 & \infty & 2 \\ 2 & 0 & \infty & 2 \\ \infty & 9 & 0 & 4 \\ 8 & 2 & 3 & 0 \end{bmatrix} \qquad (b) \begin{bmatrix} 0 & 2 & 4 & 6 \\ 2 & 0 & 1 & 2 \\ 5 & 9 & 0 & 1 \\ 9 & \infty & 2 & 0 \end{bmatrix}.$$

**7.17.** Give an example of a directed graph that contains some edges with negative costs and yet the all-pairs shortest path algorithm gives the correct distances.

**7.18.** Give an example of a directed graph that contains some edges with negative costs such that the all-pairs shortest path algorithm fails to give the correct distances.

**7.19.** Show how to modify the all-pairs shortest path algorithm so that it detects negative-weight cycles (A negative-weight cycle is a cycle whose total length is negative).

**7.20.** Prove Observation 7.2.

**7.21.** Solve the following instance of the knapsack problem. There are four items of sizes 2, 3, 5, and 6 and values 3, 4, 5, and 7, and the knapsack capacity is 11.

**7.22.** Solve the following instance of the knapsack problem. There are five items of sizes 3, 5, 7, 8 and 9 and values 4, 6, 7, 9 and 10, and the knapsack capacity is 22.

**7.23.** Explain what would happen when running the knapsack algorithm on an input in which one item has negative size.

**7.24.** Show how to modify Algorithm KNAPSACK so that it requires only $\Theta(C)$ space, where $C$ is the knapsack capacity.

**7.25.** Show how to modify Algorithm KNAPSACK so that it outputs the items packed in the knapsack as well.

**7.26.** In order to lower the prohibitive running time of the knapsack problem, which is $\Theta(nC)$, we may divide $C$ and all the $s_i$'s by a large number $K$ and take the floor. That is, we may transform the given instance into a new instance with capacity $\lfloor C/K \rfloor$ and item sizes $\lfloor s_i/K \rfloor, 1 \leq i \leq n$. Now, we apply the algorithm for the knapsack discussed in Sec. 7.6. This technique is called *scaling and rounding* (see Sec. 15.6). What will be the running time of the algorithm when applied to the new instance? Give a counterexample to show that scaling and rounding does not always result in an optimal solution to the *original* instance.

**7.27.** Another version of the knapsack problem is to let the set $U$ contain a set of *types* of items, and the objective is to fill the knapsack with any number of items of each type in order to maximize the total value without exceeding the knapsack capacity. Assume that there is an unlimited number of items of each type. More formally, let $T = \{t_1, t_2, \ldots, t_n\}$ be a set of $n$ *types* of items, and $C$ the knapsack capacity. For $1 \leq j \leq n$, let $s_j$ and $v_j$ be, respectively, the size and value of the items of type $j$. Find a set of nonnegative integers $x_1, x_2, \ldots, x_n$ such that

$$\sum_{i=1}^{n} x_i v_i$$

is maximized subject to the constraint

$$\sum_{i=1}^{n} x_i s_i \leq C.$$

$$x_1, x_2, \ldots, x_n \text{ are nonnegative integers.}$$

Note that $x_j = 0$ means that no item of the $j$th type is packed in the knapsack. Rewrite the dynamic programming algorithm for this version of the knapsack problem.

**7.28.** Solve the following instance of the version of the knapsack problem described in Exercise 7.27. There are five types of items with sizes 2, 3, 5 and 6 and values 4, 7, 9 and 11, and the knapsack capacity is 8.

**7.29.** Show how to modify the knapsack algorithm discussed in Exercise 7.27 so that it computes the number of items packed from each type.

**7.30.** Consider the *money change* problem. We have a currency system that has $n$ coins with values $v_1, v_2, \ldots, v_n$, where $v_1 = 1$, and we want to pay change of value $y$ in such a way that the total number of coins is minimized. More formally, we want to minimize the quantity

$$\sum_{i=1}^{n} x_i$$

subject to the constraint

$$\sum_{i=1}^{n} x_i v_i = y.$$

Here, $x_1, x_2, \ldots, x_n$ are nonnegative integers (so $x_i$ may be zero).

(a) Give a dynamic programming algorithm to solve this problem.

(b) What are the time and space complexities of your algorithm?

(c) Can you see the resemblance of this problem to the version of the knapsack problem discussed in Exercise 7.27? Explain.

**7.31.** Apply the algorithm in Exercise 7.30 to the instance $v_1 = 1, v_2 = 5, v_3 = 7, v_4 = 11$ and $y = 20$.

**7.32.** Let $G = (V, E)$ be a directed graph with $n$ vertices. $G$ induces a relation $R$ on the set of vertices $V$ defined by: $u\ R\ v$ if and only if there is a directed edge from $u$ to $v$, i.e., if and only if $(u, v) \in E$. Let $M_R$ be the adjacency matrix of $G$, i.e., $M_R$ is an $n \times n$ matrix satisfying $M_R[u, v] = 1$ if $(u, v) \in E$ and 0 otherwise. The *reflexive and transitive closure* of $M_R$, denoted by $M_R^*$, is defined as follows. For $u, v \in V$, if $u = v$ or there is a path in $G$ from $u$ to $v$, then $M_R^*[u, v] = 1$ and 0 otherwise. Give a dynamic programming algorithm to compute $M_R^*$ for a given directed graph. (Hint: You only need a slight modification of Floyd's algorithm for the all-pairs shortest path problem).

**7.33.** Let $G = (V, E)$ be a directed graph with $n$ vertices. Define the $n \times n$ distance matrix $D$ as follows. For $u, v \in V$, $D[u, v] = d$ if and only if the length of the shortest path from $u$ to $v$ measured in the number of edges is exactly $d$. For example, for any $v \in V$, $D[v, v] = 0$ and for any $u, v \in V$ $D[u, v] = 1$ if and only if $(u, v) \in E$. Give a dynamic programming algorithm to compute the distance matrix $D$ for a given directed graph. (Hint: Again, you only need a slight modification of Floyd's algorithm for the all-pairs shortest path problem).

**7.34.** Let $G = (V, E)$ be a directed acyclic graph (dag) with $n$ vertices. Let $s$ and $t$ be two vertices in $V$ such that the indegree of $s$ is 0 and the outdegree of $t$ is 0. Give a dynamic programming algorithm to compute a longest path in $G$ from $s$ to $t$. What is the time complexity of your algorithm?

**7.35.** Give a dynamic programming algorithm for the traveling salesman problem: Given a set of $n$ cities with their intercity distances, find a *tour* of minimum length. Here, a tour is a cycle that visits each city exactly once. What is the time complexity of your algorithm? This problem can be solved using dynamic programming in time $O(n^2 2^n)$ (see the bibliographic notes).

**7.36.** Let $P$ be a convex polygon with $n$ vertices (see Sec. 18.2). A *chord* in $P$ is a line segment that connects two nonadjacent vertices in $P$. The problem of *triangulating a convex polygon* is to partition the polygon into $n - 2$ triangles by drawing $n - 3$ chords inside $P$. Figure 7.8 shows two possible triangulations of the same convex polygon.

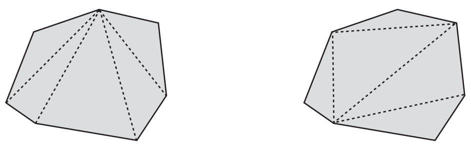

Fig. 7.8 Two triangulations of the same convex polygon.

(a) Show that the number of ways to triangulate a convex polygon with $n$ vertices is the same as the number of ways to multiply $n - 1$ matrices.

(b) A *minimum weight* triangulation is a triangulation in which the sum of the lengths of the $n - 3$ chords is minimum. Give a dynamic programming algorithm for finding a minimum weight triangulation of a convex polygon with $n$ vertices. (Hint: This problem is very similar to the matrix chain multiplication covered in Sec. 7.3).

```
Algorithm 8.6 HUFFMAN
Input: A set $C = \{c_1, c_2, \ldots, c_n\}$ of $n$ characters and their frequencies
       $\{f(c_1), f(c_2), \ldots, f(c_n)\}$.
Output: A Huffman tree (V, T) for $C$.

 1. Insert all characters into a min-heap $H$ according to their frequencies.
 2. $V \leftarrow C$;    $T = \{\}$
 3. for $j \leftarrow 1$ to $n-1$
 4.     $c \leftarrow$ DELETEMIN$(H)$
 5.     $c' \leftarrow$ DELETEMIN$(H)$
 6.     $f(v) \leftarrow f(c) + f(c')$      $\{v$ is a new node$\}$
 7.     INSERT$(H, v)$
 8.     $V = V \cup \{v\}$      $\{$Add $v$ to $V\}$
 9.     $T = T \cup \{(v, c), (v, c')\}$      $\{$Make $c$ and $c'$ children of $v$ in $T\}$
10. end while
```

## 8.6 Exercises

**8.1.** Is Algorithm LINEARSEARCH described in Sec. 1.3 a greedy algorithm? Explain.

**8.2.** Is Algorithm MAJORITY described in Sec. 5.7 a greedy algorithm? Explain.

**8.3.** This exercise is about the *money change* problem stated in Exercise 7.30. Consider a currency system that has the following coins and their values: dollar (100 cents), quarter (25 cents), dime (10 cents), nickel (5 cents) and 1-cent coins. (A unit-value coin is always required). Suppose we want to give a change of value $n$ cents in such a way that the total number of coins $n$ is minimized. Give a greedy algorithm to solve this problem.

**8.4.** Give a counterexample to show that the greedy algorithm obtained in Exercise 8.3 does not always work if we instead use coins of values 1 cent, 5 cents, 7 cents and 11 cents. Note that in this case dynamic programming can be used to find the minimum number of coins. (See Exercises 7.30 and 7.31).

**8.5.** Suppose in the money change problem of Exercise 8.3 the coin values are: $1, 2, 4, 8, 16, \ldots, 2^k$, for some positive integer $k$. Give an $O(\log n)$ algorithm to solve the problem if the value to be paid is $n < 2^{k+1}$.

**8.6.** For what denominations $\{v_1, v_2, \ldots, v_k\}, k \geq 2$, does the greedy algorithm for the money change problem stated in Exercise 7.30 always give the minimum number of coins? Prove your answer.

**8.7.** Let $G = (V, E)$ be an undirected graph. A vertex cover for $G$ is a subset

$S \subseteq V$ such that every edge in $E$ is incident to at least one vertex in $S$. Consider the following algorithm for finding a vertex cover for $G$. First, order the vertices in $V$ by decreasing order of degree. Next execute the following step until all edges are covered. Pick a vertex of highest degree that is incident to at least one edge in the remaining graph, add it to the cover, and delete all edges incident to that vertex. Show that this greedy approach does not always result in a vertex cover of minimum size.

**8.8.** Let $G = (V, E)$ be an undirected graph. A clique $C$ in $G$ is a subgraph of $G$ that is a complete graph by itself. A clique $C$ is maximum if there is no other clique $C'$ in $G$ such that the size of $C'$ is greater than the size of $C$. Consider the following method that attempts to find a maximum clique in $G$. Initially, let $C = G$. Repeat the following step until $C$ is a clique. Delete from $C$ a vertex that is not connected to every other vertex in $C$. Show that this greedy approach does not always result in a maximum clique.

**8.9.** Let $G = (V, E)$ be an undirected graph. A coloring of $G$ is an assignment of colors to the vertices in $V$ such that no two adjacent vertices have the same color. The coloring problem is to determine the minimum number of colors needed to color $G$. Consider the following greedy method that attempts to solve the coloring problem. Let the colors be $1, 2, 3, \ldots$. First, color as many vertices as possible using color 1. Next, color as many vertices as possible using color 2, and so on. Show that this greedy approach does not always color the graph using the minimum number of colors.

**8.10.** Let $A_1, A_2, \ldots, A_m$ be $m$ arrays of integers each sorted in nondecreasing order. Each array $A_j$ is of size $n_j$. Suppose we want to merge all arrays into one array $A$ using an algorithm similar to Algorithm MERGE described in Sec. 1.4. Give a greedy strategy for the order in which these arrays should be merged so that the overall number of comparisons is minimized. For example, if $m = 3$, we may merge $A_1$ with $A_2$ to obtain $A_4$ and then merge $A_3$ with $A_4$ to obtain $A$. Another alternative is to merge $A_2$ with $A_3$ to obtain $A_4$ and then merge $A_1$ with $A_4$ to obtain $A$. Yet another alternative is to merge $A_1$ with $A_3$ to obtain $A_4$ and then merge $A_2$ with $A_4$ to obtain $A$. (Hint: Give an algorithm similar to Algorithm HUFFMAN).

**8.11.** Analyze the time complexity of the algorithm in Exercise 8.10.

**8.12.** Consider the following greedy algorithm which attempts to find the distance from vertex $s$ to vertex $t$ in a directed graph $G$ with positive lengths on its edges. Starting from vertex $s$, go to the nearest vertex, say $x$. From vertex $x$, go to the nearest vertex, say $y$. Continue in this manner until you arrive at vertex $t$. Give a graph with the fewest number of vertices to show that this heuristic does not always produce the distance from $s$

to $t$. (Recall that the distance from vertex $u$ to vertex $v$ is the length of a shortest path from $u$ to $v$).

**8.13.** Apply Algorithm DIJKSTRA on the directed graph shown in Fig. 8.7. Assume that vertex 1 is the start vertex.

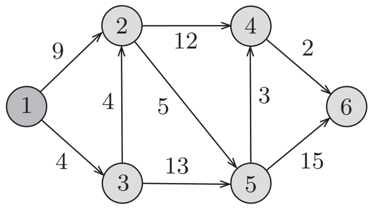

Fig. 8.7 Directed graph.

**8.14.** Is Algorithm DIJKSTRA optimal? Explain.

**8.15.** What are the merits and demerits of using the adjacency matrix representation instead of the adjacency lists in the input to Algorithm DIJKSTRA?

**8.16.** Modify Algorithm DIJKSTRA so that it finds the shortest paths in addition to their lengths.

**8.17.** Prove that the subgraph defined by the paths obtained from the modified shortest path algorithm as described in Exercise 8.16 is a tree. This tree is called the *shortest path tree*.

**8.18.** Can a directed graph have two distinct shortest path trees (see Exercise 8.17)? Prove your answer.

**8.19.** Give an example of a directed graph to show that Algorithm DIJKSTRA does not always work if some of the edges have negative weights.

**8.20.** Show that the proof of correctness of Algorithm DIJKSTRA (Lemma 8.1) does not work if some of the edges in the input graph have negative weights.

**8.21.** Let $G = (V, E)$ be a directed graph such that removing the directions from its edges results in a planar graph. What is the running time of Algorithm SHORTESTPATH when applied to $G$? Compare that to the running time when using Algorithm DIJKSTRA.

**8.22.** Let $G = (V, E)$ be a directed graph such that $m = O(n^{1.2})$, where $n = |V|$ and $m = |E|$. What changes should be made to Algorithm SHORTESTPATH so that it will run in time $O(m)$?

**8.23.** Show the result of applying Algorithm KRUSKAL to find a minimum cost spanning tree for the undirected graph shown in Fig. 8.8.

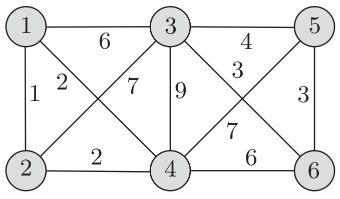

Fig. 8.8 An undirected graph.

**8.24.** Show the result of applying Algorithm PRIM to find a minimum cost spanning tree for the undirected graph shown in Fig. 8.8.

**8.25.** Let $G = (V, E)$ be an undirected graph such that $m = O(n^{1.99})$, where $n = |V|$ and $m = |E|$. Suppose you want to find a minimum cost spanning tree for $G$. Which algorithm would you choose: Algorithm PRIM or Algorithm KRUSKAL? Explain.

**8.26.** Let $e$ be an edge of minimum weight in an undirected graph $G$. Show that $e$ belongs to some minimum cost spanning tree of $G$.

**8.27.** Does Algorithm PRIM work correctly if the graph has negative weights? Prove your answer.

**8.28.** Let $G$ be an undirected weighted graph such that no two edges have the same weight. Prove that $G$ has a unique minimum cost spanning tree.

**8.29.** What is the number of spanning trees of a *complete* undirected graph $G$ with $n$ vertices? For example, the number of spanning trees of $K_3$, the complete graph with three vertices, is 3.

**8.30.** Let $G$ be a directed weighted graph such that no two edges have the same weight. Let $T$ be a shortest path tree for $G$ (see Exercise 8.17). Let $G'$ be the undirected graph obtained by removing the directions from the edges of $G$. Let $T'$ be a minimum spanning tree for $G'$. Prove or disprove that $T = T'$.

**8.31.** Use Algorithm HUFFMAN to find an optimal code for the characters a, b, c, d, e and f whose frequencies in a given text are respectively 7, 5, 3, 2, 12, 9.

**8.32.** Prove that the graph obtained in Algorithm HUFFMAN is a tree.

**8.33.** Algorithm HUFFMAN constructs the code tree in a bottom-up fashion. Is it a dynamic programming algorithm?

**8.34.** Let $B = \{b_1, b_2, \ldots, b_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$ be two sets of black and white points in the plane. Each point is represented by the pair $(x, y)$ of $x$ and $y$ coordinates. A black point $b_i = (x_i, y_i)$ dominates a white point $w_j = (x_j, y_j)$ if and only if $x_i \geq x_j$ and $y_i \geq y_j$. A *matching* between a black point $b_i$ and a white point $w_j$ is possible if

## 10.8 Exercises

**10.1.** Give an efficient algorithm to solve the decision version of the SORTING stated on page 282. What is the time complexity of your algorithm?

**10.2.** Give an efficient algorithm to solve the problem SET DISJOINTNESS stated on page 282. What is the time complexity of your algorithm?

**10.3.** Design a polynomial time algorithm for the problem 2-COLORING defined on page 282. (Hint: Color the first vertex white, all adjacent vertices black, etc).

**10.4.** Design a polynomial time algorithm for the problem 2-SAT defined on page 282.

**10.5.** Let $I$ be an instance of the problem COLORING, and let $s$ be a claimed solution to $I$. Describe a deterministic algorithm to test whether $s$ is a solution to $I$.

**10.6.** Design a nondeterministic algorithm to solve the problem SATISFIABILITY.

**10.7.** Design a nondeterministic algorithm to solve the problem TRAVELING SALESMAN.

**10.8.** Show that $\text{P} \subseteq \text{NP}$.

**10.9.** Let $\Pi_1$ and $\Pi_2$ be two problems such that $\Pi_1 \propto_{poly} \Pi_2$ . Suppose that problem $\Pi_2$ can be solved in $O(n^k)$ time and the reduction can be done in $O(n^j)$ time. Show that problem $\Pi_1$ can be solved in $O(n^{jk})$ time.

**10.10.** Given that the Hamiltonian cycle problem for undirected graphs is NP-complete, show that the Hamiltonian cycle problem for directed graphs is also NP-complete.

**10.11.** Show that the problem BIN PACKING is NP-complete, assuming that the problem PARTITION is NP-complete.

**10.12.** Let $\Pi_1$ and $\Pi_2$ be two NP-complete problems. Prove or disprove that $\Pi_1 \propto_{poly} \Pi_2$ .

**10.13.** Give a polynomial time algorithm to find a clique of size $k$ in a given undirected graph $G = (V, E)$ with $n$ vertices. Here $k$ is a *fixed* positive integer. Does this contradict the fact that the problem CLIQUE is NP-complete? Explain.

**10.14.** Consider the following instance of SATISFIABILITY:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2})$$

(a) Following the reduction method from SATISFIABILITY to CLIQUE, transform the above formula into an instance of CLIQUE for which the answer is *yes* if and only if the the above formula is satisfiable.

(b) Find a clique of size 4 in your graph and convert it into a satisfying assignment for the formula given above.

**10.15.** Consider the formula $f$ given in Exercise 10.14.

(a) Following the reduction method from SATISFIABILITY to VERTEX COVER, transform $f$ into an instance of VERTEX COVER for which the answer is *yes* if and only if $f$ is satisfiable.

(b) Find a vertex cover in your graph and convert it into a satisfying assignment for $f$.

**10.16.** The NP-completeness of the problem CLIQUE was shown by reducing SATISFIABILITY to it. Give a simpler reduction from VERTEX COVER to CLIQUE.

**10.17.** Show that any cover of a clique of size $n$ must have exactly $n-1$ vertices.

**10.18.** Show that if one can devise a polynomial time algorithm for the problem SATISFIABILITY then NP = P (see Exercise 10.9).

**10.19.** In Chapter 7 it was shown that the problem KNAPSACK can be solved in time $\Theta(nC)$, where $n$ is the number of items and $C$ is the knapsack capacity. However, it was mentioned in this chapter that it is NP-complete. Is there any contradiction? Explain.

**10.20.** When showing that an optimization problem is not harder than its decision problem version, it was justified by using binary search and an algorithm for the decision problem in order to solve the optimization version. Will the justification still be valid if linear search is used instead of binary search? Explain. (Hint: Consider the problem TRAVELING SALESMAN).

**10.21.** Prove that if an NP-complete problem $\Pi$ is shown to be solvable in polynomial time, then NP = P (see Exercises 10.9 and 10.18).

**10.22.** Prove that NP = P if and only if for some NP-complete problem $\Pi$, $\Pi \in$ P.

**10.23.** Is the problem LONGEST PATH NP-complete when the path is not restricted to be simple? Prove your answer.

**10.24.** Is the problem LONGEST PATH NP-complete when restricted to directed acyclic graphs? Prove your answer. (See Exercises 10.23 and 7.34).

**10.25.** Show that the problem of finding a shortest *simple* path between two vertices $s$ and $t$ in a directed or undirected graph is NP-complete if the weights are allowed to be negative.

**10.26.** Show that the problem SET COVER is NP-complete by reducing the problem VERTEX COVER to it.

**10.27.** Show that the problem 3-SAT is NP-complete.

**10.28.** Show that the problem 3-COLORING is NP-complete.

**10.29.** Compare the difficulty of the problem TAUTOLOGY to SATISFIABILITY. What does this imply about the difficulty of the class co-NP.

**10.30.** Prove Theorem 10.7.

## 10.9 Bibliographic notes

The study of NP-completeness started with two papers. The first was the seminal paper of Cook (1971) in which the problem SATISFIABILITY was the first problem shown to be NP-complete. The second was Karp (1972) in which a list of 24 problems were shown to be NP-complete. Both Stephen Cook and Richard Karp have won the ACM Turing awards and their Turing award lectures were published in Cook (1983) and Karp (1986). Garey and Johnson (1979) provides comprehensive coverage of the theory of NP-completeness and covers the four basic complexity classes introduced in this chapter. Their book contains the proof that SATISFIABILITY is NP-complete and a list of several hundred NP-complete problems. One of the most famous of the open problems to be resolved is LINEAR PROGRAMMING. This problem has been proven to be solvable in polynomial time using the ellipsoid method (Khachiyan (1979)). It has received much attention, although its practical significance is yet to be determined. An introduction to the theory of NP-completeness can also be found in Aho, Hopcroft and Ullman (1974) and Hopcroft and Ullman (1979).

(3) Stop and accept.

We observe that checking whether $(G, k)$ is in COLORING is implemented by asking the oracle COLORING and is answered in one step, by assumption. So, the algorithm presented above is clearly polynomial time bounded, since it needs at most two steps to either accept or reject. It follows that CHROMATIC NUMBER is in $\Delta_2^p = \mathrm{P}^{\mathrm{NP}}$.

**Example 11.7** MINIMUM EQUIVALENT EXPRESSION. Given a well-formed boolean expression $E$ and a nonnegative integer $k$, is there a well-formed boolean expression $E'$ that contains $k$ or fewer occurrences of literals such that $E'$ is equivalent to $E$ (i.e. $E'$ if and only if $E$)?

MINIMUM EQUIVALENT EXPRESSION does not appear to be in $\Delta_2^p$. It is not obvious whether an oracle for a problem in NP can be used to solve MINIMUM EQUIVALENT EXPRESSION in *deterministic* polynomial time. However, this problem can be solved in *nondeterministic* polynomial time using an oracle for SATISFIABILITY. The algorithm is as follows

(1) Guess a boolean expression $E'$ containing $k$ or fewer occurrences of literals.
(2) Use SATISFIABILITY to determine whether $\neg((E' \rightarrow E) \wedge (E \rightarrow E'))$ is satisfiable.
(3) If it is *not* satisfiable then stop and accept, otherwise stop and reject.

The correctness of the above algorithm follows from the fact that a well-formed formula $E$ is *not* satisfiable if and only if its negation is a tautology. Thus, since we want ($E'$ if and only if $E$) to be a tautology, we only need to check whether

$$\neg((E' \rightarrow E) \wedge (E \rightarrow E'))$$

is *not* satisfiable. As to the time needed, Step 1, generating $E'$, can easily be accomplished in polynomial time using a nondeterministic algorithm. Step 2, querying the SATISFIABILITY oracle, is done in one step. It follows that MINIMUM EQUIVALENT EXPRESSION is in $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$.

## 11.10 Exercises

**11.1.** Show that the language in Example 11.1 is in DTIME($n$). (Hint: Use a 2-tape Turing machine).

**11.2.** Show that the language $L = \{ww \mid w \in \{a, b\}^+\}$ is in LOGSPACE by constructing a log space bounded off-line Turing machine that recognizes $L$. Here $\{a, b\}^+$ denotes all nonempty strings over the alphabet $\{a, b\}$.

**11.3.** Consider the following decision problem of sorting: Given a sequence of $n$ distinct positive integers between 1 and $n$, are they sorted in increasing order? Show that this problem is in

(a) DTIME($n \log n$).

(b) LOGSPACE.

**11.4.** Give an algorithm to solve the problem $k$-CLIQUE defined in Example 11.5. Use the $O$-notation to express the time complexity of your algorithm.

**11.5.** Show that the problem $k$-CLIQUE defined in Example 11.5 is in LOGSPACE.

**11.6.** Show that the problem GAP is in LOGSPACE if the graph is undirected.

**11.7.** Consider the following decision problem of the selection problem. Given an array $A[1..n]$ of integers, an integer $x$ and an integer $k, 1 \leq k \leq n$, is the $k$th smallest element in $A$ equal to $x$? Show that this problem is in LOGSPACE.

**11.8.** Let $A$ be an $n \times n$ matrix. Show that computing $A^2$ is in LOGSPACE. How about computing $A^k$ for an arbitrary $k \geq 3$, where $k$ is part of the input?

**11.9.** Show that the problem 2-SAT described on page 282 is in NLOGSPACE. Conclude that it is in P.

**11.10.** Show that all finite sets are in LOGSPACE.

**11.11.** Show that the family of sets accepted by finite state automata is a *proper* subset of LOGSPACE. (Hint: The language $\{a^n b^n \mid n \geq 1\}$ is not a ccepted by any finite state automaton, but it is in LOGSPACE.

**11.12.** Show that if $T_1$ and $T_2$ are two time-constructible functions, then so are $T_1 + T_2$, $T_1 T_2$ and $2^{T_1}$.

**11.13.** Prove Corollary 11.5.

**11.14.** Show that if NSPACE($n$) $\subseteq$ NP then NP = NSPACE. Conclude that NSPACE($n$) $\neq$ NP.

**11.15.** Show that if LOGSPACE = NLOGSPACE, then for every space constructible function $S(n) \geq \log n$, DSPACE($S(n)$) = NSPACE($S(n)$).

**11.16.** Describe a log space reduction from the set $L = \{www \mid w \in \{a,b\}^+\}$ to the set $L' = \{ww \mid w \in \{a,b\}^+\}$. That is, show that $L \propto_{log} L'$.

**11.17.** Show that the relation $\propto_{poly}$ is transitive. That is, if $\Pi \propto_{poly} \Pi'$ and $\Pi' \propto_{poly} \Pi''$, then $\Pi \propto_{poly} \Pi''$.

**11.18.** Show that the relation $\propto_{log}$ is transitive. That is, if $\Pi \propto_{log} \Pi'$ and $\Pi' \propto_{log} \Pi''$, then $\Pi \propto_{log} \Pi''$.

**11.19.** The problems 2-COLORING and 2-SAT were defined in Sec. 10.2. Show that 2-COLORING is log space reducible to 2-SAT. (Hint: Let $G = (V, E)$. Let the boolean variable $x_v$ correspond to vertex $v$ for each vertex $v \in V$, and for each edge $(u, v) \in E$ construct the two clauses $(x_u \vee x_v)$ and $(\neg x_u \vee \neg x_v)$).

**11.20.** A graph $V = (G, E)$ is bipartite if and only if $V$ can be partitioned into two sets $X$ and $Y$ such that all edges in $E$ are of the form $(x, y)$ with $x \in X$ and $y \in Y$. Equivalently, $G$ is bipartite if and only if it does not contain odd-length cycles (see Sec. 3.3). Show that deciding whether a graph is bipartite is log space reducible to the problem 2-COLORING described in Sec. 10.2.

**11.21.** Show that for any $k \geq 1$, DTIME($n^k$) is not closed under polynomial time reductions.

**11.22.** Show that, for any $k \geq 1$, the class DSPACE($\log^k n$) is closed under log space reductions.

**11.23.** A set S is *linear time reducible* to a set $T$, denoted by $S \propto_n T$, if there exists a function $f$ that can be computed in linear time (that is, $f(x)$ can be computed in $c|x|$ steps, for all input strings $x$, where $c$ is some constant $> 0$) such that

$$\forall x \; x \in S \text{ if and only if } f(x) \in T.$$

Show that if $S \propto_n T$ and $T$ is in DTIME($n^k$), then $S$ is in DTIME($n^k$). That is, DTIME($n^k$) ($k \geq 1$) is closed under linear time reducibility.

**11.24.** Suppose that $k$ in Exercise 11.5 is not fixed, that is, $k$ is part of the input. Will the problem still be in LOGSPACE? Explain.

**11.25.** Show that the class NLOGSPACE is closed under complementation. Conclude that the complement of the problem GAP is NLOGSPACE-complete.

**11.26.** Show that the problem GAP remains NLOGSPACE-complete even if the graph is acyclic.

**11.27.** Show that the problem 2-SAT described in Sec. 10.2 is complete for the class NLOGSPACE under log space reduction (see Exercise 11.9). (Hint: Reduce the complement of the problem GAP to it. Let $G = (V, E)$ be a directed acyclic graph. GAP is NLOGSPACE-complete even if the graph is acyclic (Exercise 11.26). By Exercise 11.25, the complement of the problem GAP is NLOGSPACE-complete. Associate with each vertex $v$ in $V$ a boolean variable $x_v$. Associate with each edge $(u, v) \in E$ the clause $(\neg x_u \vee x_v)$, and add the clauses $(x_s)$ for the start vertex and $(\neg x_t)$ for the goal vertex $t$. Prove that 2-SAT is satisfiable if and only if there is no path from $s$ to $t$).

**11.28.** Define the class

$$\text{POLYLOGSPACE} = \bigcup_{k\geq 1} \text{DSPACE}(\log^k n).$$

Show that there is no set that is complete for the class POLYLOGSPACE. (Hint: The class DSPACE($\log^k n$) is closed under log space reduction).

**11.29.** Prove that PSPACE $\subseteq$ P if and only if PSPACE $\subseteq$ PSPACE($n$). (Hint: Use padding argument).

**11.30.** Does there exist a problem that is complete for the class DTIME($n$) under log space reduction? Prove your answer.

**11.31.** Use the fact that there is an NP-complete problem to show that there is no problem that is complete for the class NTIME($n^k$) under log space reduction for any $k \geq 1$.

**11.32.** Let $\mathcal{L}$ be a class that is closed under complementation and let the set $L$ (that is not necessarily in $\mathcal{L}$) be such that

$$\forall L' \in \mathcal{L}\ L' \propto L.$$

Show that

$$\forall L'' \in \text{co-}\mathcal{L}\ L'' \propto \overline{L}.$$

**11.33.** Show that for any class of languages $\mathcal{L}$, if $L$ is complete for the class $\mathcal{L}$, then $\overline{L}$ is complete for the class co-$\mathcal{L}$.

**11.34.** Show that NLOGSPACE is strictly contained in PSPACE.

**11.35.** Show that DEXT $\neq$ PSPACE. (Hint: Show that DEXT is not closed under $\propto_{poly}$).

**11.36.** Prove

(a) Theorem 11.15(1).
(b) Theorem 11.15(2).
(c) Theorem 11.15(3).

**11.37.** Prove

(a) Theorem 11.16(1).
(b) Theorem 11.16(2).

**11.38.** Prove

(a) Theorem 11.17(1).
(b) Theorem 11.17(2).

**11.39.** Prove Theorem 11.18.

**11.40.** Show that the problem PATH SYSTEM ACCESSIBILITY$\in$ P.

**11.41.** Show that polynomial time Turing reduction as defined on page 325 implies polynomial time transformation as defined in Sec. 11.7. Is the converse true? Explain.

**11.42.** Consider the MAX-CLIQUE problem defined as follows. Given a graph $G = (V, E)$ and a positive integer $k$, decide whether the maximal complete subgraph of $G$ is of size $k$. Show that MAX-CLIQUE is in $\Delta_2^p$.

**11.43.** Prove that $\Sigma_1^p = \text{NP}$.

**11.44.** Show that if $\Sigma_k^p \subseteq \Pi_k^p$, then $\Sigma_k^p = \Pi_k^p$.

**11.45.** Let PLOGSPACE be the class of sets accepted by a parallel model of computation using logarithmic space. Show that PLOGSPACE = P.

## 11.11 Bibliographic notes

Part of the material in this chapter is based on Sudborough(1982). Other references include Balcazar, Diaz and Gabarro (1988,1990), Bovet and Crescenzi (1994), Garey and Johnson (1979), Hopcroft and Ullman (1979) and Papadimitriou(1994). The book by Bovet and Crescenzi (1994) provides a good introduction to the field of computational complexity. The first attempt to make a systematic approach to computational complexity was made by Rabin(1960). The study of time and space complexity can be said to begin with Hartmanis and Stearns(1965), Stearns, Hartmanis and Lewis(1965) and Lewis, Stearns and Harmanis(1965). This work contains most of the basic theorems of complexity classes and time and space hierarchy. Theorem 11.4 is due to Savitch (1970). Extensive research in this field emerged and enormous number of papers have been published since then. For comments about NP-complete problems, see the bibliographic notes of Chapter 10. PSPACE-complete problems were first studied in Karp (1972) including CSG RECOGNITION and LBA ACCEPTANCE. QUANTIFIED BOOLEAN FORMULAS was shown to be PSPACE-complete in Stockmeyer and Meyer (1973) and Stockmeyer (1974). The *linear bounded automata problem*, which predates the NP = P question, is the problem of deciding whether nondeterministic LBA's are equivalent to deterministic LBA's, that is whether NSPACE($n$) = DSPACE($n$).

NLOGSPACE-complete problems were studied by Savitch (1970), Sudborough (1975a,b), Springsteel (1976), Jones (1975), and Jones, Lien and Lasser (1976). The NLOGSPACE-completeness of the GRAPH ACCESSIBILITY problem (GAP) was proven in Jones (1975).

because it caused the greatest increase in the lower bound of the right subtree. This heuristic is useful because it is faster to find the solution by following the left edges, which reduce the dimension as opposed to the right edges which merely add a new $\infty$ and probably more zeros. However, we did not use this heuristic when splitting at the node containing matrix $C$. It is left as an exercise to find the optimal solution with fewer node splittings.

From the above example, it seems that the heap is an ideal data structure to use in order to expand the node with the least cost (or maximum cost in case of maximization). Although branch-and-bound algorithms are generally complicated and hard to program, they proved to be efficient in practice.

## 13.6 Exercises

**13.1.** Let $k$-COLORING be a generalization of the 3-COLORING problem presented in Sec. 13.2. How many nodes are generated by its corresponding backtracking algorithm in the worst case?

**13.2.** Consider the algorithm for 3-COLORING presented in Sec. 13.2. Give an efficient algorithm to test whether a vector corresponding to a 3-coloring of a graph is legal.

**13.3.** Consider the algorithm for 3-COLORING presented in Sec. 13.2. Explain how to efficiently test whether the current vector is partial throughout the execution of the algorithm.

**13.4.** Let Algorithm n-queens be a generalization of Algorithm 4-QUEENS presented in Sec. 13.3 for the case of an $n \times n$ chessboard. How many nodes are generated by Algorithm n-queens in the worst case?

**13.5.** Show that two queens placed at positions $x_i$ and $x_j$ are in the same diagonal if and only if

$$x_i - x_j = i - j \text{ or } x_i - x_j = j - i.$$

**13.6.** Give a recursive algorithm for the 8-QUEENS problem.

**13.7.** Does the $n$-queen problem have a solution for every value of $n \geq 4$? Prove your answer.

**13.8.** Modify Algorithm 4-QUEENS so that it reduces the search space from $4^4$ to 4! as described in Sec. 13.3.

**13.9.** Design a backtracking algorithm to generate all permutations of the numbers $1, 2, \ldots, n$.

**13.10.** Design a backtracking algorithm to generate all $2^n$ subsets of the numbers $1, 2, \ldots, n$.

**13.11.** Write a backtracking algorithm to solve the *knight tour problem*: Given an $8 \times 8$ chessboard, decide if it is possible for a knight placed at a certain position of the board to visit every square of the board exactly once and return to its start position.

**13.12.** Write a backtracking algorithm to solve the following variant of the PARTITION problem (see Example 13.3): Given $n$ positive integers $X = \{x_1, x_2, \ldots, x_n\}$ and a positive integer $y$, does there exist a subset $Y \subseteq X$ whose elements sum up to $y$?

**13.13.** Give a backtracking algorithm to solve the HAMILTONIAN CYCLE problem: Given an undirected graph $G = (V, E)$, determine whether it contains a simple cycle that visits each vertex exactly once.

**13.14.** Consider the KNAPSACK problem defined in Sec. 7.6. It was shown that using dynamic programming, the problem can be solved in time $\Theta(nC)$, where $n$ is the number of items and $C$ is the knapsack capacity.

(a) Give a backtracking algorithm to solve the knapsack problem.

(b) Which technique is more efficient to solve the knapsack problem: backtracking or dynamic programming? Explain.

**13.15.** Give a backtracking algorithm to solve the money change problem defined in Exercise 7.30.

**13.16.** Apply the algorithm in Exercise 13.15 for the money change problem on the instance in Exercise 7.31.

**13.17.** Give a backtracking algorithm to solve the *assignment problem* defined as follows. Given $n$ employees to be assigned to $n$ jobs such that the cost of assigning the $i$th person to the $j$th job is $c_{i,j}$, find an assignment that minimizes the total cost. Assume that the cost is nonnegative, that is, $c_{i,j} \geq 0$ for $1 \leq i, j \leq n$.

**13.18.** Modify the solution of the instance of the TRAVELING SALESMAN problem given in Sec. 13.5 so that it results in fewer node splittings.

**13.19.** Apply the branch-and-bound algorithm for the TRAVELING SALESMAN problem discussed in Sec. 13.5 on the instance

$$\begin{bmatrix} \infty & 5 & 2 & 10 \\ 2 & \infty & 5 & 12 \\ 3 & 7 & \infty & 5 \\ 8 & 2 & 4 & \infty \end{bmatrix}.$$

**13.20.** Consider again the KNAPSACK problem defined in Sec. 7.6. Use branch and bound and a suitable lower bound to solve the instance of this problem in Example 7.6.

**13.21.** Carry out a branch-and-bound procedure to solve the following instance of the assignment problem defined in Exercise 13.17. There are four employees and four jobs. The cost function is represented by the matrix below. In this matrix, row $i$ corresponds to the $i$th employee, and column $j$ corresponds to the $j$th job.

$$\begin{bmatrix} 3 & 5 & 2 & 4 \\ 6 & 7 & 5 & 3 \\ 3 & 7 & 4 & 5 \\ 8 & 5 & 4 & 6 \end{bmatrix}.$$

## 13.7 Bibliographic notes

There are several books that cover backtracking in some detail. These include Brassard and Bratley (1988), Horowitz and Sahni (1978), Reingold, Nievergelt and Deo (1977). It is also described in Golomb and Brumert (1965). Techniques for analyzing its efficiency are given in Knuth (1975). The recursive form of backtracking was used by Tarjan (1972) in various graph algorithms. Branch-and-bound techniques have been successfully used in optimization problems since the late 1950s. Many of the diverse applications are outlined in the survey paper by Lawler and Wood (1966). The approach to solve the TRAVELING SALESMAN problem in this chapter is due to Little, Murty, Sweeney and Karel (1963). Another technique to solve the TRAVELING SALESMAN problem is described in the survey paper by Bellmore and Nemhauser (1968).