1.19 Solutions

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.

(a) 6. (b) 5. (c) 6. (d) 6.

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.

(a) 10. (b) 10. (c) 1. (d) 11.

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.

Similar to Fig. 1.1.

1.4. Show that the height of the decision tree for binary search is $\lfloor \log n \rfloor$.

Proof is by induction on n, the number of nodes.
Basis step: If $n = 1$, then there is only one element, and one node in the decision tree, and $\lfloor \log 1 \rfloor = 0$, which is the height of the tree.
Induction step: Suppose the hypothesis holds for all trees with less than n nodes. We show that it also holds for n. Let T be a decision tree with n nodes. The root r of T corresponds to the first comparison. Let T_l and T_h be the two subtrees of r, where T_l corresponds to subsequent comparisons in $A[1..\lfloor n/2 \rfloor]$, and T_h corresponds to subsequent comparisons in $A[\lfloor n/2 \rfloor + 1..n]$. By induction, the height of T_l is $\lfloor \log(n/2) \rfloor$, and the height of T_h is $\lfloor \log(n/2) \rfloor$. Hence, the height of T is $\max\{\lfloor \log(n/2) \rfloor, \lfloor \log(n/2) \rfloor\} + 1 = \lfloor \log(n/2) \rfloor + 1 = \lfloor \log n \rfloor - 1 + 1 = \lfloor \log n \rfloor$. Thus, the hypothesis holds for n, and hence the height of the decision tree for binary search is $\lfloor \log n \rfloor$ for all $n \geq 1$.

[§]Some of the solutions in this chapter were contributed by Faisal Alvi.

1.5. 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?

At each iteration of Algorithm SELECTIONSORT, the array has the situation shown in Table 1.2.

Table 1.2 Algorithm SELECTIONSORT. Exercise 1.5.

Iteration	Array
0	45,33,24,45,12,12,24,12
1	12,33,24,45,45,12,24,12
2	12,12,24,45,45,33,24,12
3	12,12,12,45,45,33,24,24
4	12,12,12,24,45,33,45,24
5	12,12,12 ,24,24,45,33,45
6	12,12,12,24,24,33,45,45
7	12,12,12,24,24,33,45,45

Note that in iterations 6 and 7 there is no interchange in the array. No. of comparisons = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 7*6/2 = 21.

1.6. Consider modifying Algorithm SELECTIONSORT as shown in Algorithm MODSELECTIONSORT.

Algorithm 1.24 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?

In Algorithm MODSELECTIONSORT, instead of maintaining a variable k, which stores the position of the minimum element of A we immediately interchange the minimum element with $A[i]$.

(a) Minimum number of element assignments $= 0$. This minimum is achieved when the array is already sorted in ascending order.

(b) Maximum number of element assignments $= 3n(n-1)/2$. This maximum is achieved when the array is already sorted in descending order.

1.7. Illustrate the operation of Algorithm INSERTIONSORT on the array

$$\boxed{30}\boxed{12}\boxed{13}\boxed{13}\boxed{44}\boxed{12}\boxed{25}\boxed{13}\,.$$

How many comparisons are performed by the algorithm?

At each iteration of Algorithm INSERTIONSORT the array has the situation shown in Table 1.3 (the sorted part of the array is separated by |)

Table 1.3 Algorithm INSERTIONSORT. Exercise 1.7.

Iteration	Array	# comparisons
0	30 | 12 13 13 44 12 25 13	0
1	12 30 | 13 13 44 12 25 13	1
2	12 13 30 | 13 44 12 25 13	2
3	12 13 13 30 | 44 12 25 13	2
4	12 13 13 30 44 | 12 25 13	1
5	12 12 13 13 30 44 | 25 13	5
6	12 12 13 13 25 30 44 | 13	3
7	12 12 13 13 13 25 30 44	4

No. of comparisons $= 1 + 2 + 2 + 1 + 5 + 3 + 4 = 18$.

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

4	3	12	5	6	7	2	9

?

At each iteration of Algorithm INSERTIONSORT the array has the situation shown in Table 1.4 (the sorted part of the array is separated by |)

Table 1.4 Algorithm INSERTIONSORT. Exercise 1.8.

Iteration	Array	# comparisons
0	4 | 3 12 5 6 7 2 9	0
1	3 4 | 12 5 6 7 2 9	1
2	3 4 12 | 5 6 7 2 9	1
3	3 4 5 12 | 6 7 2 9	2
4	3 4 5 6 12 | 7 2 9	2
5	3 4 5 6 7 12 | 2 9	2
6	2 3 4 5 6 7 12 | 9	6
7	2 3 4 5 6 7 9 12	2

No. of comparisons = 1 + 1 + 2 + 2 + 2 + 6 + 2 = 16.

1.9. Prove Observation 1.4.

The number of element comparisons performed by Algorithm INSERTIONSORT is between $n - 1$ and $n(n-1)/2$. The number of element assignments is equal to the number of element comparisons plus $n - 1$.

Proof. For the number of element comparisons, see Sec. 1.6.
For the number of element assignments, note that each iteration of the inner while loop includes one comparison and one assignment. But if the while loop test fails, there is one element comparison but no element assignment. In other words, if there are k element comparisons, then there are $k-1$ element assignments. If the outer for loop is executed $n-1$ times, then there are $(n-1)(k-1)$ element assignments and $(n-1)k$ element comparisons within the while loop itself. Furthermore, in each iteration of the **for** loop, 2 assignments

(Steps 2 and 8) take place. Therefore in $n-1$ iterations, $2(n-1)$ assignments take place.
Hence, the number of element comparisons $= (n-1)k$, and number of element assignments $= 2(n-1)+(n-1)(k-1) = n-1+(n-1)k$, which is equal to $n-1+$ number of element comparisons. $\square$

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

Algorithm INSERTIONSORT is more efficient because the number of element comparisons in Algorithm INSERTIONSORT is less than or equal to those in Algorithm SELECTIONSORT. For large records, element assignments are more costly than element comparisons. So, Algorithm SELECTIONSORT may be preferable since it performs the least number of element assignments.

1.11. Illustrate the operation of Algorithm BOTTOMUPSORT on the array

$$A[1..16] = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 11&12&1&5&15&3&4&10&7&2&16&9&8&14&13&6\\\hline\end{array}.$$

How many comparisons are performed by the algorithm?

Similar to the examples in Figs. 1.3 and 1.4.

1.12. Illustrate the operation of Algorithm BOTTOMUPSORT on the array

$$A[1..11] = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 2&17&19&5&13&11&4&8&15&12&7\\\hline\end{array}.$$

How many comparisons are performed by the algorithm?

Similar to the examples in Figs. 1.3 and 1.4.

1.13. 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.

The minimum number of element comparisons is $(n\log n)/2 = (8\log 8)/2 = 12$, for the array: $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 1&2&3&4&5&6&7&8\\\hline\end{array}$.
The maximum number of element comparisons is $n\log n - n + 1 = 17$, for the array: $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 10&15&9&20&5&25&3&40\\\hline\end{array}$.

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

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

1.15. Express the following functions in terms of the Θ-notation.
(a) $2n + 3\log^{100} n$.
(b) $7n^3 + 1000n \log n + 3n$.
(c) $3n^{1.5} + (\sqrt{n})^3 \log n$.
(d) $2^n + 100^n + n!$.

(a) $\Theta(n)$. (b) $\Theta(n^3)$. (c) $\Theta(n^{1.5} \log n)$. (d) $\Theta(n!)$.

1.16. Express the following functions in terms of the Θ-notation.
(a) $18n^3 + \log n^8$.
(b) $(n^3 + n)/(n + 5)$.
(c) $\log^2 n + \sqrt{n} + \log\log n$.
(d) $n!/2^n + n^n$.

(a) $\Theta(n^3)$. (b) $\Theta(n^2)$. (c) $\Theta(\sqrt{n})$. (d) $\Theta(n^n)$.

1.17. Consider the sorting algorithm shown below, which is called BUBBLESORT.
(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 Ω notations.

Algorithm 1.25 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 $i \leq n-1$ 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
```

(f) Can the running time of the algorithm be expressed in terms of the Θ-notation? Explain.

(a) Algorithm BUBBLESORT works as follows: In each iteration, adjacent elements are compared and interchanged from the end of the array to the beginning. In this way, the smallest element "bubbles" to the first position, the 2nd smallest element "bubbles" to the second position in the array and so on.

(b) The minimum number of element comparisons $= n-1$. This happens when the array is sorted in increasing order.

(c) The maximum number of element comparisons $= n(n-1)/2$, when the array is sorted in decreasing order.

(d) The minimum number of assignments $= 0$, when the array is sorted.

(e) Assuming that each interchange involves 3 assignments, the maximum number of assignments will be achieved when each comparison operation results in an assignment. The maximum number of assignments $= 3n(n-1)/2$.

(f) Assume that element comparison is the basic operation. In terms of upper bound, the running time is $O(n^2)$. In terms of lower bound, the running time is $\Omega(n)$.

(g) No. Because worst case and best case times belong to different complexity classes.

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

Two such functions are: $f(n) = n + \sin^2(n)$ and $g(n) = n + \cos^2(n)$. It can be empirically verified that $f(n)$ and $g(n)$ satisfy the stated conditions.

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

$f(n)$ is $O(g(n))$ if for all $x \geq x_0$, there exists a constant $c > 0$ such that $f(n) \leq cg(n)$. Here, $x \leq cx \sin x$ or $c \sin x \geq 1$.
There is no constant $c > 0$ satisfying the above inequality since $\sin x$ fluctuates between -1 and +1 for $x \geq x_0$. It follows that $x \neq O(x \sin x)$.

1.20. 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 $\Theta(n^{k+1})$.

$\sum_{j=1}^{n} j^k = n^k + (n-1)^k + \ldots + 2^k + 1^k < n \times n^k$. Hence, $\sum_{j=1}^{n} j^k = O(n^{k+1})$. On the other hand,

$$\sum_{j=1}^{n} j^k = \underbrace{n^k + (n-1)^k + \ldots + (\lfloor n/2 \rfloor)^k}_{n/2 \text{ terms}} + \ldots + 2^k + 1^k.$$

Hence, $\sum_{j=1}^{n} j^k > (n/2) \times (n/2)^k = n^{k+1}/2^{k+1} = \Omega(n^{k+1})$. It follows that $\sum_{j=1}^{n} j^k = \Theta(n^{k+1})$.

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

Write $f(n) = \frac{1}{n} + \frac{1}{n} f(n)$. Then, $f(n)(1 - \frac{1}{n}) = \frac{1}{n}$, or $f(n) = \frac{1}{n-1}$. It follows that $f(n) = \Theta(1/n)$.

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

$\lim_{n\to\infty} \frac{n^{100}}{2^n} = \lim_{n\to\infty} \frac{2^{100 \log n}}{2^n} = 0$. Hence, $n^{100} = o(2^n)$, which means $n^{100} = O(2^n)$, but $2^n \neq O(n^{100})$. See Exercise 1.44 for the more general case.

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

$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$. Hence, $2^n = o(3^n)$, which means $2^n \neq \Theta(3^n)$.

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

$$\begin{aligned}\lim_{n\to\infty} \frac{n!}{n^n} &= \lim_{n\to\infty} \frac{n}{n}\frac{n-1}{n}\frac{n-2}{n}\ldots\frac{1}{n} \\ &= \lim_{n\to\infty} 1 \times \left(1-\frac{1}{n}\right) \times \left(1-\frac{2}{n}\right) \times \ldots \times \frac{2}{n} \times \frac{1}{n} \\ &= 1 \times 1 \times 1 \times \ldots \times 0 \times 0 = 0\end{aligned}$$

Hence, $n! = o(n^n)$, which means $n! \neq \Theta(n^n)$.

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

$$\lim_{n\to\infty} \frac{2^{n^2}}{2^{n^3}} = \lim_{n\to\infty} \left(\frac{1}{2^{n^3-n^2}}\right) = \lim_{n\to\infty} \left(\frac{1}{2^{n^2}}\right) \lim_{n\to\infty} \left(\frac{1}{2^{n-1}}\right) = 0.$$

Hence, $2^{n^2} = o(2^{n^3})$, which means $2^{n^2} \neq \Theta(2^{n^3})$.

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

$f(n) = O(1)$ means $f(n)$ is bounded above by a constant, e.g. $\frac{1}{n}, 5, n^{-3}\log n$. On the other hand, $f(n) = \Theta(1)$ means $f(n)$ is bounded above and below by a constant, e.g. $\frac{n+5}{n+6}, 100$.

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

Consider the case when $n = 2^k$, so that $\lfloor \log n \rfloor = \log n = k$. It is easy to see that $k! = \Omega(2^k)$ (Consider $\lim_{n\to\infty} \frac{k}{2}\frac{k-1}{2}\frac{k-2}{2}\ldots\frac{1}{2} \neq 0$). Hence, $k! = \Omega(2^k) = \Omega(n)$.
The above argument can be genralized to $\lfloor \log n \rfloor$ since $\log n - 1 < \lfloor \log n \rfloor \leq \log n$.

1.28. 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.

No, since n^2 and $100n^2$ are both quadratic and belong to the same complexity class.

1.29. 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}, $n!$.

$(1/2)^n \prec 5 \prec \log \log n \prec \log n^{100} \prec \log^2 n \prec n^{1/100} \prec \sqrt{n} \prec n \log n \prec (\sqrt{n})^n \prec n! \prec 2^{n^2}$.

1.30. 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 Θ notations is more appropriate to measure the number of multiplications? Explain.

(b) Which one of the O and Θ notations is more appropriate to measure the number of element tests? Explain.

(a) The O-notation is better since the minimum number of multiplications is zero and the maximum is n.

(b) The Θ-notation is better since the number of element tests is exactly n.

1.31. Give a more efficient algorithm than the one given in Example 1.22. What is the time complexity of your algorithm?

Algorithm COUNT1MOD computes for each perfect square l between 1 and n the sum $\sum_{i=1}^{l} i$. Its running time is $\Theta(n)$.
A much more efficient method is given by Algorithm COUNT1MOD2 below. Its running time is $\Theta(k) = \Theta(\sqrt{n})$.

1.32. Consider Algorithm COUNT6 whose input is a positive integer n.

(a) How many times Step 6 is executed?

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

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

(a) Let $r = \lfloor \log n \rfloor$. Then, number of times Step 6 is executed is

$$\sum_{i=1}^{r} \sum_{j=i}^{i+5} \sum_{k=1}^{i^2} 1 = \sum_{i=1}^{r} \sum_{j=i}^{i+5} i^2 = \sum_{i=1}^{r} 6i^2 = r(r+1)(2r+1).$$

Algorithm 1.26 COUNT1MOD
Input: $n = k^2$ for some integer k.
Output: $\sum_{i=1}^{l} i$ for each perfect square l between 1 and n.

```
 1. $k \leftarrow \sqrt{n}$
 2. $sum[1] \leftarrow 1$
 3. for $j \leftarrow 2$ to $k$
 4.     $s \leftarrow 0$
 5.     for $r \leftarrow (j-1)^2 + 1$ to $j^2$
 6.         $s \leftarrow s + r$
 7.     end for
 8.     $sum[j] \leftarrow sum[j-1] + s$
 9. end for
10. return $sum[1..k]$
```

Algorithm 1.27 COUNT1MOD2
Input: $n = k^2$ for some integer k.
Output: $\sum_{i=1}^{l} i$ for each perfect square l between 1 and n.

```
1. $k \leftarrow \sqrt{n}$
2. for $j \leftarrow 1$ to $k$
3.     $l \leftarrow j^2$.
4.     $sum[j] \leftarrow l(l+1)/2$
5. end for
6. return $sum[1..k]$
```

Algorithm 1.28 COUNT6

```
1. comment: Exercise 1.32
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
```

(b) The Θ-notation since the bound is tight.

(c) $\Theta(r^3) = \Theta(\log^3 n)$.

1.33. Consider Algorithm COUNT7 whose input is a positive integer n.

Algorithm 1.29 COUNT7

```
1. comment: Exercise 1.33
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 time complexity of the algorithm expressed in the O-notation?

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

(d) Which one of the O and Θ notations is more appropriate to express the time complexity of the algorithm? Explain briefly.

(a) For $n = 2^k$, each time the inner **while** loop executes for a maximum of k times, the outer **for** loop executes n times. So, the maximum number of times is $nk = n \log n$.

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

(c) $\Omega(n)$ for an input of the form $2^k - 1$.

(d) The O-notation is more suitable since the upper and lower bounds do not match.

1.34. Consider Algorithm COUNT8 whose input is a positive integer n.

(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

Algorithm 1.30 COUNT8

```
comment: Exercise 1.34
count $\leftarrow$ 0
for $i \leftarrow 1$ to $n$
    $j \leftarrow \lfloor n/3 \rfloor$
    while $j \geq 1$
        for $k \leftarrow 1$ to $i$
            count $\leftarrow$ count $+ 1$
        end for
        if $j$ is even then $j \leftarrow 0$ else $j \leftarrow \lfloor j/3 \rfloor$
    end while
end for
```

Ω-notation?

(e) Which one of the O and Θ notations is more appropriate to express the time complexity of the algorithm? Explain briefly.

(a) The analysis is complicated, but it has been empirically found that if the input is $n = 2^k$, for $0 \leq k \leq 51$, the algorithm executes Step 7 the maximum number of times when $k = 16$.

(b) Let $n = 3^r$. Then, the maximum number of times Step 7 is executed is

$$\sum_{i=1}^{n} \sum_{l=1}^{r} \sum_{k=1}^{i} 1 = \sum_{i=1}^{n} ir = \frac{n(n+1)}{2} \log_3 n.$$

(c) $O(n^2 \log n)$.

(d) $\Omega(n^2)$ (for example, when the input is $6m$, where m is any positive integer).

(e) The O-notation is more suitable since the upper and lower bounds do not match.

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

(a) Scan the array from left to right, inspecting each element, and return the maximum and minimum.

(b) Sort the array and return $A[1]$ and $A[n]$.

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

Return the median of the first three elements.

1.37. 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?

One way to solve the element uniqueness problem is to compare each element with every other element and determine if any two are equal. This will take $n(n-1)/2$ comparisons in the worst case which means $O(n^2)$ time complexity. A better way is to sort the array in time $\Theta(n \log n)$. Then, by comparing adjacent elements only (A[1] and A[2], A[2] and A[3], ...) we can determine in $n-1$ comparisons if any two elements are equal. The time complexity is $\Theta(n \log n)$.

1.38. 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) $O(n)$.

(a) One way is to compute $a_n x^n, a_{n-1} x^{n-1}$, etc. separately. This can be done by multiplying x with itself n times and then multiplying the result by a_n to compute $a_n x^n$. This takes n multiplications. Similarly $a_{n-1} x^{n-1}$ will take $n-1$ multiplications. Together the entire generation of terms would take at least $\sum_{i=n}^{1} i = n(n+1)/2 = \Omega(n^2)$ operations.

(b) We can use Horner's method, which evaluates a polynomial by additions and multiplications simultaneously. Compute $a_n x$ by a single multiplication. Add a_{n-1} to give the term $a_n x + a_{n-1}$. Multiply the result by x again to give $a_n x^2 + a_{n-1} x$, etc. In other words, we compute

$$a_0 + x(a_1 + \ldots + x(a_{n-2} + x(a_{n-1} + a_n x)) \ldots).$$

Thus, in a total of n multiplications (and additions) we'll generate the polynomial, which is a maximum of $O(n)$ operations.

1.39. 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?

A good way to partition an array into two halves such that the difference between the elements in the lower half and those in the upper half of the elements is maximum is by simply sorting the array. This operation would take time $\Theta(n \log n)$.

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

It seems the only way to achieve minimum partitioning is to check all possible partitions. There are $\binom{n}{n/2}$ partitions of $n/2$ elements each in an array of n elements.

1.41. 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 $\gcd(m, n)$.

(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

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

Algorithm 1.31 EUCLID
Input: Two positive integers m and n.
Output: $\gcd(m, n)$.

```
1. comment: Exercise 1.41
2. repeat
3.     $r \leftarrow n \bmod m$
4.     $n \leftarrow m$
5.     $m \leftarrow r$
6. until $r = 0$
7. return $n$
```

(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 Θ-notation? Explain.

(a) No, it doesn't matter because if $n < m$ then n and m will get interchanged in the first iteration.

(b) Suppose that $\gcd(m, n) = x$. Then, n mod m is the smallest positive difference obtained by subtracting multiples of m from n. In other words, $n \bmod m = n - pm$ for some p such that $n - (p + 1)m < 0$. So, if x divides n and m, then it must also divide $n - pm = n \bmod m$. In other words,

$$\begin{aligned}
\gcd(n, m) &= \gcd(m, n \bmod m) \\
&= \gcd(n \bmod m, m \bmod (n \bmod m)) \\
&\ \ \vdots \\
&= \gcd(x, 0) \\
&= x.
\end{aligned}$$

(c) In the Euclidean algorithm, every two iterations reduce the integer n to n mod m. To achieve maximum number of iterations, it is desirable that n must be reduced as little as possible. This can be achieved when in each iteration m is subtracted only once. In other words, $p = 1$ for $n \bmod m = n - pm$. Furthermore, the gcd must also be 1 in order to ensure the maximum number of iterations. In the fibonacci sequence, $f_n \bmod f_{n-1} = f_{n-2}$ since $f_n - f_{n-1} = f_{n-2}$ and $f_n - 2f_{n-1} < 0$, so that

$$\gcd(f_n, f_{n-1}) = \gcd(f_{n-1}, f_{n-2}) = \ldots = \gcd(1, 0) = 1.$$

It takes n iterations of the Euclidean algorithm to find $\gcd(f_n, f_{n-1})$. Since in each iteration, the quotient of division of f_n and f_{n-1} is always 1 and the gcd is also 1, it follows that finding $\gcd(f_n, f_{n-1})$ takes maximum time in the Euclidean algorithm.

(d) Assume that $n > m$. Then, after the first two iterations, the new value of n is $n - m \times \lfloor n/m \rfloor$. If $m > n/2$, then $n - m \times \lfloor n/m \rfloor = n - m < n/2$. On the other hand, if $m < n/2$, then if $n = pm + r$, where r is the remainder after division and $p \geq 2$, then $\lfloor n/m \rfloor = p$, and hence $n - m \times \lfloor n/m \rfloor = n - m \times p = pm + r - pm = r$ which is less than m, which in turn is less than $n/2$. So, for all values of m, n reduces by at least one half in each two consecutive iterations. In other words, $T(n/2) \leq T(n/2) + c$, which means that $T(n) = O(\log n)$.

(e) No, because time may vary form input to input.

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

Measured in terms of its input size (the number of digits in n), the algorithm is linear because n consists of $\lfloor \log_{10} n \rfloor + 1$ digits. Assuming $k = \lfloor \log_{10} n \rfloor + 1$, we have the time complexity of the algorithm as $O(\log n)$, which is $O(k)$. Hence the algorithm is linear in terms of its input size.

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

Follows from the fact that $\lim_{n \to \infty} \frac{\log^c n}{n} = 0$.

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

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

Follows from the fact that $\lim_{n\to\infty} \frac{n^c}{d^n} = 0$.

1.45. 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 1.3.

(a) Let $n = 2^k$. Expanding the recurrence, we have

$$\begin{aligned} f(n) &= 4^k + n\sum_{j=0}^{k-1} 2^j \\ &= n^2 + n\frac{2^k - 1}{2 - 1} \\ &= n^2 + n(n-1) \\ &= 2n^2 - n \\ &= \Theta(n^2). \end{aligned}$$

(b) $f(n) = \Theta(n^2)$.

1.46. 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 1.3.

(a) Let $n = 3^k$. Expanding the recurrence yields

$$\begin{aligned} f(n) &= 5^{\log_3 n} + n\sum_{j=0}^{k-1} \left(\frac{5}{3}\right)^j \\ &= n^{\log_3 5} + n\frac{(5/3)^k - 1}{(5/3) - 1} \\ &= n^{\log_3 5} + \frac{5^k - n}{2/3} \end{aligned}$$

$$
\begin{aligned}
&= n^{\log_3 5} + \frac{3}{2}(n^{\log_3 5} - n) \\
&= \frac{5}{2}n^{\log_3 5} - \frac{3}{2}n \\
&= \Theta(n^{\log_3 5}).
\end{aligned}
$$

(b) $f(n) = \Theta(n^{\log_3 5})$.

1.47. 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 1.3.

(a) Let $n = 3^k$. Expanding the recurrence, we have

$$
\begin{aligned}
f(n) &= 9^k + n\sum_{j=0}^{k-1}\left(\frac{9}{9}\right)^j \\
&= n^{\log_3 9} + n^2 \log_3 n \\
&= n^2 + n^2 \log_3 n \\
&= \Theta(n^2 \log n).
\end{aligned}
$$

(b) $f(n) = \Theta(n^2 \log n)$.

1.48. 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 1.3.

(a) Let $n = 4^r$. Expanding the recurrence, we have

$$
\begin{aligned}
f(n) &= 2^r f(n/4^r) + \sum_{j=0}^{r-1}\sqrt{n} \\
&= 2^r + \sqrt{n}\log_4 n \\
&= \sqrt{n} + \sqrt{n}\log_4 n
\end{aligned}
$$

$$= \Theta(\sqrt{n} \log n).$$

(b) $f(n) = \Theta(\sqrt{n} \log n)$.

1.49. 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.

Let cn^x be an upper bound for $f(n)$, that is, $f(n) \leq cn^x$. Then, substituting in the recurrence, we obtain

$$f(n) \leq c\left(\frac{n}{2}\right)^x + c\left(\frac{3n}{4}\right)^x.$$

We want this quantity to be $\leq cn^x$. Hence,

$$c\left(\frac{n}{2}\right)^x + c\left(\frac{3n}{4}\right)^x \leq cn^x.$$

Simplifying (dividing by cn^x) yields

$$\left(\frac{1}{2}\right)^x + \left(\frac{3}{4}\right)^x \leq 1,$$

or

$$2^x + 3^x \leq 4^x. \tag{1.1}$$

This inequality is satisfied if $x \approx 1.51$. Hence, $f(n) = O(n^x)$ for x satisfying Eq. 1.1.

1.50. 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.

Let $f(n) = cn \log n$. Then,

$$f(n) \leq c\left(\frac{n}{4}\right) \log\left(\frac{n}{4}\right) + c\left(\frac{3n}{4}\right) \log\left(\frac{3n}{4}\right) + n$$

$$\begin{aligned} &= \left(\frac{c}{4}\right) n\log n + \left(\frac{3c}{4}\right) n\log n - \left(\frac{cn}{4}\right)\times 2 \\ &\quad -\left(\frac{3cn}{4}\right)\log\left(\frac{4}{3}\right) + n \\ &= cn\log n - cn\beta + n, \end{aligned}$$

where $\beta = \frac{1}{2}+\frac{3}{4}\log\frac{4}{3}$. If we let $f(n) \le cn\log n - cn\beta + n \le cn\log n$, then $cn\beta \ge n$ or $c\beta \ge 1$. Hence, $c \ge \frac{1}{\beta} = \frac{1}{0.81}$, which implies $c \ge 1.23$. It follows that $f(x) = O(n\log n)$.

1.51. Use the substitution method to find a lower bound for the recurrence in Exercise 1.49. Express the solution using the Ω-notation.

The solution is similar to that of Exercise 1.49. If we reverse inequalities in the solution to Exercise 1.49, we obtain $f(n) = \Omega(n^x)$ for all x satisfying $2^x + 3^x \ge 4^x$, e.g. $f(n) = \Omega(n^{1.5})$.

1.52. Use the substitution method to find a lower bound for the recurrence in Exercise 1.50. Express the solution using the Ω-notation.

The solution is similar to that of Exercise 1.50. So, $f(x) = \Omega(n\log n)$.

1.53. Use the substitution method to solve the recurrence

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

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

Let $f(n) = cn^2$. Then,

$$\begin{aligned} f(n) &= 2c\left(\frac{n}{2}\right)^2 + n^2 \\ &= \frac{1}{2}cn^2 + n^2 \end{aligned}$$

If we let $\frac{1}{2}cn^2 + n^2 = cn^2$, and solve for c, we obtain $\frac{c}{2} + 1 = c$ or $c = 2$. Hence, $f(n) = 2n^2 = \Theta(n^2)$.

1.54. Let

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

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.

They are equal. $f(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1$, and

$$\begin{aligned} g(n) &= 1 + 2(1 + 2(1 + 2(\ldots 1 + 2(1)\ldots))) \\ &= 1 + 2 + 2^2 + 2^3 + \ldots + 2^{\log n} \\ &= n + \frac{n}{2} + \frac{n}{4} + \ldots + 2 + 1. \end{aligned}$$

1.55. 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 $f(n)$.

Since $n = 2^{2^k}$, $\frac{n}{2} = 2^{2^k - 1}$ and $\sqrt{n} = 2^{2^{k-1}}$. Hence,

$$f(2^{2^k}) = f(2^{2^k - 1}) + 2^{2^{k-1}}.$$

Let $m = 2^k$, and define

$$g(2^k) = g(2^k - 1) + 2^{2^{k-1}}, \; k \geq 1; \quad g(2^0) = 2.$$

Then,

$$g(m) = g(m-1) + 2^{m/2}, \; m \geq 2; \quad g(1) = 2.$$

The solution to this recurrence is

$$\begin{aligned} g(m) &= 2^{m/2} + 2^{(m-1)/2} + 2^{(m-2)/2} + \ldots + 2^{2/2} + g(1) \\ &= (\sqrt{2})^m + (\sqrt{2})^{m-1} + (\sqrt{2})^{m-2} + \ldots + (\sqrt{2})^2 + 2 \\ &= \left(\frac{(\sqrt{2})^{m+1} - 1}{\sqrt{2} - 1} - \sqrt{2} - 1 \right) + 2 \\ &= \Theta((\sqrt{2})^m) = \Theta(2^{2^{k-1}}) = \Theta(\sqrt{n}). \end{aligned}$$

Hence, $f(n) = \Theta(\sqrt{n})$.

1.56. 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 $f(n)$.

Since $n = 2^{2^k}$, $\sqrt{n} = 2^{2^{k-1}}$. Hence,

$$f(2^{2^k}) = 2f(2^{2^{k-1}}) + 2^{2^k}.$$

Let $g(k) = f(2^{2^k})$. Then,

$$g(k) = 2g(k-1) + 2^{2^k},\ k \geq 1; \quad g(0) = 1.$$

Let $g(k) = 2^k h(k)$. Then,

$$2^k h(k) = 2 \times 2^{k-1} h(k-1) + 2^{2^k},$$

or

$$h(k) = h(k-1) + 2^{2^k}/2^k,\ k \geq 1; \quad h(0) = 1.$$

The solution to this recurrence is

$$h(k) = \sum_{i=1}^{k} \frac{2^{2^i}}{2^i} + 1.$$

Substituting, we obtain

$$\begin{aligned}
f(n) &= 2^k h(k) \\
&= 2^k \left(\sum_{i=1}^{k} \frac{2^{2^i}}{2^i} + 1 \right) \\
&\leq 2^k \left(\sum_{i=1}^{k} \frac{2^{2^k}}{2^k} + 1 \right) \\
&= \log n \left(\sum_{i=1}^{\log \log n} \frac{n}{\log n} + 1 \right) \\
&= n \log \log n + \log n.
\end{aligned}$$

Hence, $f(n) = O(n \log \log n)$.

1.57. 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 $g(n) = n^{1-\epsilon}$, $0 < \epsilon < 1$.

Let $n = 2^k$, and assume without loss of generality that $g(n) = n^{\epsilon}$, $0 < \epsilon < 1$. By Lemma 1.1,

$$f(n) = \left(1 + \frac{2^{\epsilon}}{2 - 2^{\epsilon}}\right) n - \left(\frac{2^{\epsilon}}{2 - 2^{\epsilon}}\right) n^{\epsilon}.$$

That is, $f(n) = O(n)$.

2.10 Solutions

2.1. Write an algorithm to delete an element x, if it exists, from a doubly-linked list L. Assume that the variable *head* points to the first element in the list and the functions $pred(y)$ and $next(y)$ return the predecessor and successor of node y, respectively.

See Algorithm DELETEFROMLIST below.

Algorithm 2.1 DELETEFROMLIST
Input: A doubly-linked list L and an element x.

Output: Delete x from L.

```
1. Search for $x$. Let $p$ be a pointer to $x$ if it exists, else let $p \leftarrow nil$
2. if $p = nil$ then exit
3. if $p = head$ then set $head \leftarrow next(p)$ and $pred(next(p)) \leftarrow nil$
4. else if $next(p) = nil$ then set $next(pred(p)) \leftarrow nil$
5. else
6.     Set $next(pred(p)) = next(p)$
7.     Set $pred(next(p)) = pred(p)$
8. end if
```

2.2. Give an algorithm to test whether a list has a repeated element.

Let L be the list, and i and j two pointers. Scan L from left to right, and for each element scanned $L(i)$, traverse the list using pointer j starting at $next(L(i))$. Return **true** whenever $L(i) = L(j)$.

2.3. Rewrite Algorithm INSERTIONSORT so that its input is a doubly linked list of n elements instead of an array. Will the time complexity change? Is the new algorithm more efficient?

Algorithm INSERTIONSORT using linked lists instead of arrays is shown below as Algorithm INSERTIONSORTLIST. In the algorithm, i and j are pointers to elements in the input list L. The time complexity is still $O(n^2)$. The algorithm is not more efficient than Algorithm INSERTIONSORT, but it has less element assignments.

2.4. A polynomial of the form $p(x) = a_1x^{b_1} + a_2x^{b_2} + \ldots + a_nx^{b_n}$, where $b_1 > b_2 > \ldots > b_n \geq 0$, can be represented by a linked list in which each record has three fields for a_i, b_i and the link to the

Algorithm 2.2 INSERTIONSORTLIST
Input: A list L of n elements.
Output: L sorted in nondecreasing order.

```
1.  $i \leftarrow head(L)$
2.  while $next(i) \neq nil$
3.      $i \leftarrow next(i)$
4.      $j \leftarrow i$
5.      $x \leftarrow L(i)$
6.      Delete $x$ from $L$.
7.      while $pred(j) \neq nil$ and $L(pred(j)) > x$
8.          $j \leftarrow pred(j)$
9.      end while
10.     Insert $x$ at $L(j)$
11. end while
```

next record. Give an algorithm to add two polynomials using this representation. What is the running time of your algorithm?

Let the two polynomials be $p(x) = a_1x^{b_1} + a_2x^{b_2} + \ldots + a_nx^{b_n}$, represented by list L_1, and $q(x) = c_1x^{d_1} + c_2x^{d_2} + \ldots + c_nx^{d_m}$ represented by L_2. Scan L_1 and L_2 in parallel stopping whenever $b_i = d_j$. When $b_i = d_j$, append the term $e_ix^{b_i}$ to the output list, where $e_i = a_i + c_j$. Also, add $a_ix^{b_i}$ and $c_jx^{d_j}$ if they do not have matching terms. The running time is $\Theta(n + m)$, where n and m are the sizes of L_1 and L_2, respectively.

2.5. Give the adjacency matrix and adjacency list representations of the graph shown in Fig. 2.5.

Similar to Fig. 2.3.

2.6. Describe an algorithm to insert and delete edges in the adjacency list representation for
(a) a directed graph.
(b) an undirected graph.

(a) To insert edge (i, j), go to the pointer $head(i)$, which is the head of the list corresponding to vertex i, and add the element consisting of the label for vertex j.
(b) In the case of an undirected graph, repeat the above procedure

by adding element i in the list corresponding to vertex j.

2.7. Let S_1 be a stack containing n elements. Give an algorithm to sort the elements in S_1 so that the smallest element is on top of the stack after sorting. Assume you are allowed to use another stack S_2 as a temporary storage. What is the time complexity of your algorithm?

See Exercise 4.34.

2.8. What if you are allowed to use two stacks S_2 and S_3 as a temporary storage in Exercise 2.7?

See Exercise 4.35.

2.9. Let G be a directed graph with n vertices and m edges. When is it the case that the adjacency matrix representation is more efficient than the adjacency lists representation? Explain.

The adjacency matrix representation is more efficient than the adjacency lists representation in the case of dense graphs with number of edges in the order of $\Omega(n^2)$.

2.10. Prove that a graph is bipartite if and only if it has no odd-length cycles.

Let G be a graph. G is 2-colorable means that the vertices of G can be colored using two colors such that no two adjacent vertices have the same color. It is well-known that G is bipartite if and only if it is 2-colorable. But then the proof is immediate since G is 2-colorable if and only if it has no odd-length cycles.

2.11. Draw *the* almost-complete binary tree with
(a) 10 nodes.
(b) 19 nodes.

Similar to Fig. 2.8.

2.12. Prove Observation 2.1.

By induction on j. For the basis step, the root is the only node at level 0, and the number of nodes at level 0 is $1 \leq 2^0$.
For the induction step, assume the number of nodes on level $j-1$ is at most 2^{j-1}. Since each node has a maximum degree of 2, the maximum number of nodes at level j is 2 times the maximum number of nodes on level $j-1$, or $2 \times 2^{j-1} = 2^j$.

2.13. Prove Observation 2.2.

By Exercise 2.12, the number of nodes in a binary tree of height h is at most $\sum_{j=0}^{h} 2^j = 2^{h+1} - 1$.

2.14. Prove Observation 2.4.

Let h be the height of the almost-complete binary tree with n nodes. Then, $2^h \leq n \leq 2^{h+1} - 1$, or $2^h + 1 \leq n + 1 \leq 2^{h+1}$. Taking logs, we obtain $\log(2^h + 1) \leq \log(n+1) \leq h + 1$. Since $2^h < 2^h + 1$, we obtain $h < \log(2^h + 1)$, so that $h < \log(2^h+1) \leq \log(n+1) \leq h+1$, or $h < \log(n+1) \leq h + 1$. This implies that $\lceil \log(n+1) \rceil = h + 1$ or $h = \lceil \log(n+1) \rceil - 1$. But $\lceil \log(n+1) \rceil - 1 = \lfloor \log n \rfloor$. It follows that $h = \lfloor \log n \rfloor$.

2.15. Prove Observation 2.3.

The height of a binary tree with n nodes is minimum when it is a complete or almost-complete, that is, $\lfloor \log n \rfloor$ (Exercise 2.14). It is maximum when it is degenerate, that is, of height $n - 1$.

2.16. Prove Observation 2.5.

Let n and m be the number of nodes and edges in a full binary tree T, respectively. Think of each edge as having two ends: an edge top touching the parent, and an edge bottom touching the child. Then, # edge tops $= m =$ # edge bottoms. The number of edge bottoms is exactly the number of nodes minus one, since each node except the root lies at the bottom of exactly one edge. So # edge bottoms in $T = n - 1$. Also, each internal node in T is at the top of exactly two edges. Therefore, we have $2 \times$ # internal nodes

= # edge tops. Putting these equalities together, we have

$$\begin{aligned} 2 \times \#\text{ internal nodes} &= \#\text{ edge tops} \\ &= \#\text{ edge bottoms} \\ &= n - 1 \\ &= \#\text{ leaves} + \#\text{ internal nodes} - 1. \end{aligned}$$

Consequently, # internal nodes = # leaves - 1.

2.17. Show how to list the elements stored in a binary tree in level order. For example, in Fig 2.11, the output should be a, b, c, d, e, f, g. Hint: Use a queue.

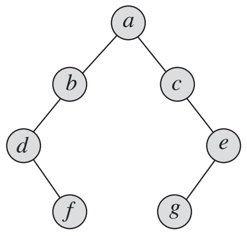

Fig. 2.11 Exercise 2.17.

This is essentially breadth-first search. After visiting a vertex, visit its children (if any). The algorithm is shown as Algorithm LEVELTRAVERSAL below. Q is a queue that is initially empty.

Algorithm 2.3 LEVELTRAVERSAL
Input: A tree T rooted at vertex v.

Output: List of vertices in level order.

```
1. $Q \leftarrow \{v\}$
2. **while** $Q \neq \{\}$
3.     $v \leftarrow Pop(Q)$
4.     **output** $v$
5.     **for** each child $w$ of $v$
6.         Push($w, Q$)
7.     **end for**
8. **end while**
```

2.18. Is a tree a bipartite graph? Prove your answer (see Exercise 2.10).

A tree is a bipartite graph since it has no odd-length cycles.

2.19. Let T be a nonempty binary search tree. Give an algorithm to
(a) return the minimum element stored in T.
(b) return the maximum element stored in T.

(a) See Algorithm MINBST below.

```
Algorithm 2.4 MINBST
Input: A binary search tree $T$.
Output: The minimum element stored in $T$
  1. Let $T'$ be the left subtree of $T$.
  2. if $T'$ is empty then return the element stored in the root.
  3. else return MINBST($T'$)
```

(b) Similar to Algorithm MINBST above. Change left to right and minimum to maximum.

2.20. Let T be a nonempty binary search tree. Give an algorithm to list all the elements in T in increasing order. What is the time complexity of your algorithm?

See Algorithm LISTBST below. The time complexity is $\Theta(n)$.

```
Algorithm 2.5 LISTBST
Input: A binary search tree $T$.
Output: The list of all elements stored in $T$
  1. Let $T_1$ be the left subtree of $T$ and $T_2$ be the right subtree of $T$.
  2. if $T_1$ is not empty then LISTBST($T_1$)
  3. output the element stored at the root
  4. if $T_2$ is not empty then LISTBST($T_2$)
```

2.21. Let T be a nonempty binary search tree. Give an algorithm to delete an element x from T, if it exists. What is the time complexity of your algorithm?

First, search for x, so assume it is found. It is easy to delete x if it is a leaf or if it has only one child; just change pointers. So, assume that x has two children. Let y be the predecessor of x, that is, the largest element among the keys in the left subtree of x. It is important to note that y cannot have a right child, since otherwise it would not have the largest key in that subtree. So, interchange x with y, and delete x, which now has at most one child. Consider, for instance, the tree shown in Fig. 2.12 (a), which is the left tree in Fig. 2.9. Fig. 2.12 (b) shows the result of deleting $x = 6$ from the tree shown in part (a) of the figure. For the running time, see Exercise 2.23.

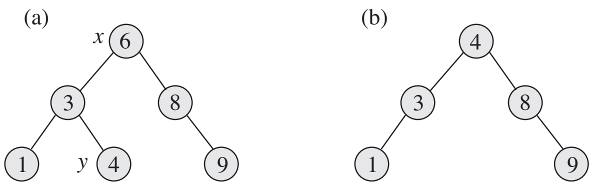

Fig. 2.12 Deletion in a binary search tree.

2.22. Let T be binary search tree. Give an algorithm to insert an element x in its proper position in T. What is the time complexity of your algorithm?

First, search for x. If it is found, then the insertion will be aborted, as there should be no duplicates in the tree. So, assume that x is not in the tree. In this case, searching for x will end unsuccessfully at a leaf node, say y. x can then be inserted as a right or left child of y depending on the value of x relative to y. For the running time, see Exercise 2.23.

2.23. What is the time complexity of deletion and insertion in a binary search tree? Explain.

The running time of deletion and insertion in a binary search tree depends on the shape of the tree. It is proportional to the height of the tree. If the tree is balanced, then the running time is $\Theta(\log n)$. In the worst case, however, the running time is $\Theta(n)$, as the tree may be degenerate and hence the height is $\Omega(n)$.

2.24. When discussing the time complexity of an operation in a binary search tree, which of the O and Θ notations is more appropriate? Explain.

The O-notation is more appropriate as discussed in Exercise 2.23.

3.7 Solutions

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

If an ordered list is used, then insertion is expensive, as it may require searching a large portion of the elements to find the proper location for insertion; it costs $O(n)$. So, insertion of n elements takes $O(n^2)$ time. Deletion takes $O(1)$ time. Finding the maximum takes $O(1)$ time.

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

If a regular queue is used, then insertion takes constant time. However, finding the largest (or smallest) element is expensive, as this requires searching the entire queue, that is, $O(n)$ for each search.

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

(a) Heap. (b) Heap. (c) Heap
(d) Heap. (e) Not a heap.

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

a) Second largest key: One of the children of the root, that is, $H[2]$ or $H[3]$.
(b) Third largest key: Either in second or third levels. That is, in $H[2..7]$
(c) Minimum key: One of the leaf nodes. That is, one of
$H[\lfloor n/2\rfloor + 1], H[\lfloor n/2\rfloor + 2], \ldots, H[n]$.

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

Algorithm HEAP below tests whether an input array A is a heap in $\Theta(n)$ time. The algorithm searches for an element in the upper half of the array that is smaller than (one of) its children. If such an element is found, the array is not a heap.

Algorithm 3.8 HEAP
Input: An array $A[1..n]$ of n elements.
Output: **true** if array A is a heap, and **false** otherwise.

```
1. for $j \leftarrow 1$ to $\lfloor n/2 \rfloor$
2.     if $A[2j] > A[j]$ then return false
3.     if $2j+1 \leq n$ and $A[2j+1] > A[j]$ then return false
4. end for
5. return true
```

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

Deletion is more costly, as it involves the sift-down operation, which is more costly than sift-up. It requires two comparisons per iteration. Sift-up requires one comparisons per iteration.

3.7. Let H be the heap shown in Fig. 3.1. Show the heap that results from

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

(a)

20	11	9	10	5	4	5	3	7

(b)

20	19	9	10	17	4	5	3	7	5	11

3.8. Show the heap (in both tree and array representation) that results from deleting the maximum key in the heap shown in Fig. 3.4(e).

Resulting heap is

26	17	13	10	11	8	7	4	3

. The tree is shown in Fig. 3.9.

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

The minimum is stored in one of the leaf nodes. So, $\lceil n/2 \rceil - 1$ comparisons are both sufficient and necessary.

3.10. 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

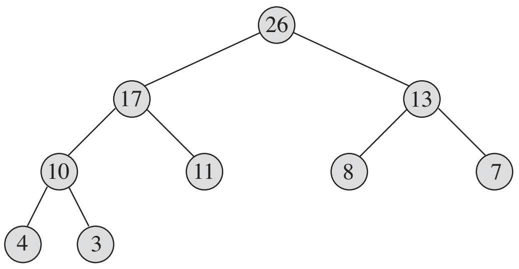

Fig. 3.9 A heap for Execise 3.8.

in T. Then, if x is greater than y, h_x cannot be less than h_y. (See Sec. 2.5 for the definition of node height).

Not true. Consider the heap $\boxed{6}\boxed{5}\boxed{2}\boxed{4}\boxed{3}\boxed{1}$. $4 > 2$, but the height of 4 is less than the height of 2.

3.11. Illustrate the operation of Algorithm MAKEHEAP on the array

3	7	2	1	9	8	6	4

.

Similar to the example shown in Fig. 3.4.

3.12. Show the steps of transforming the following array into a heap

1	4	3	2	5	7	6	8

.

Similar to the example shown in Fig. 3.4.

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

(a) How many calls to Algorithm 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.

(a) There are $\lfloor n/2 \rfloor = 9$ calls to Algorithm SIFT-DOWN.

(b) Number of iterations in Algorithm SIFT-DOWN is at most

$$2^h \sum_{i=1}^{h} (i/2^i) = 2^4 \sum_{i=1}^{4} (i/2^i) = 26,$$

which in turn is at most the number of element interchanges. The exact value is given by the sum of the numbers in the table below, which is 16 (The nodes are numbered left to right, top to bottom).

Node	9	8	7	6	5	4	3	2	1
Iterations	1	1	1	1	1	2	2	3	4

(c) Let A be the array consisting of the numbers $1, 2, \ldots, 19$ in this order. Then, A requires the above maximum number of element interchanges.

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

4	5	2	9	8	7	1	3

.

The steps are illustrated using the following arrays.

Makeheap:	9	8	7	5	4	2	1	3
Swap:	3	8	7	5	4	2	1	9
Sift-down:	8	5	7	3	4	2	1	9
Swap:	1	5	7	3	4	2	8	9
Sift-down:	7	5	2	3	4	1	8	9
Swap:	1	5	2	3	4	7	8	9
Sift-down:	5	4	2	3	1	7	8	9
Swap:	1	4	2	3	5	7	8	9
Sift-down:	4	3	2	1	5	7	8	9
Swap:	1	3	2	4	5	7	8	9
Sift-down:	3	1	2	4	5	7	8	9
Swap:	2	1	3	4	5	7	8	9
Sift-down:	2	1	3	4	5	7	8	9

Swap:	1	2	3	4	5	7	8	9
Sorted:	1	2	3	4	5	7	8	9

3.15. 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.

Insertion of the jth element costs $\Theta(\log j)$ in the worst case. Hence, the total number of operations in the worst case is

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

3.16. Illustrate the operation of the algorithm in Exercise 3.15 on the array

6	9	2	7	1	8	4	3

.

Insert 6:	6							
Insert 9:	9	6						
Insert 2:	9	6	2					
Insert 7:	9	7	2	6				
Insert 1:	9	7	2	6	1			
Insert 8:	9	7	8	6	1	2		
Insert 4:	9	7	8	6	1	2	4	
Insert 3:	9	7	8	6	1	2	4	3

3.17. Explain the behavior of Algorithm HEAPSORT when the input array is already sorted in
(a) increasing order.
(b) decreasing order.

(a) If the input array is already sorted in increasing order, the call to Algorithm MAKEHEAP costs the maximum number of element comparisons. After that, the algorithm performs

$\Theta(n \log n)$ element comparisons using Algorithm SIFT-DOWN. If all elements are identical, the running time is $\Theta(n)$.

(b) If the input array is already sorted in decreasing order, the call to Algorithm MAKEHEAP costs the minimum number of element comparisons. After that, the algorithm performs $\Theta(n \log n)$ element comparisons using Algorithm SIFT-DOWN.

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

If the keys to the left of the root are strictly smaller than the root and the keys to the right of the root are strictly larger than the root (as defined in Sec. 2.6.2), then it seems the only heaps that are also binary search trees are the trees consisting of one element and two elements. Thus, for example, the tree shown in Fig. 3.10 is a heap and binary search tree.

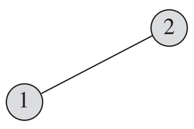

Fig. 3.10 A heap and binary search tree.

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

Let the two heaps be A and B. Append B to A to form array C with $2n$ elements. Next, apply Algorithm MAKEHEAP on C. The time complexity is $\Theta(n)$.

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

First, we compute the maximum number of element comparisons performed by the **for** loop in Step 2 of the algorithm. It is easy to see that the number of comparisons done by Algorithm SIFT-DOWN on a heap of size n is at most $2\lfloor \log n \rfloor$. Hence, by Eq. A.18 (p. 811), the total number of comparisons performed by the calls to

Algorithm SIFT-DOWN is at most

$$2\sum_{j=2}^{n}\log(j-1) = 2\sum_{j=1}^{n-1}\log j \le 2n\log n - 2n\log e.$$

By Theorem 3.1, the number of comparisons done by Algorithm MAKEHEAP is $< 4n$. Hence, the overall number of comparisons is at most

$$2n\lfloor\log n\rfloor + (4 - 2\log e)n \le 2n\log n - 1.11n.$$

To find the minimum number of element comparisons, note that there are at least $1 \times (n-1)$ sift-down iterations. Therefore, the total number of comparisons done by the sift-down algorithm is at least $2(n-1)$. By Theorem 3.1, the number of comparisons done by Algorithm MAKEHEAP is $> n-1$. Hence, the overall number of comparisons is at least $3(n-1)$.

3.21. 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 $d \ge 2$. Rewrite Algorithm SIFT-UP for the case of d-heaps. What is its time complexity?

The algorithm is shown in Procedure SIFTUP-d below. Note that in a d-ary heap A, the parent of $A[i]$ is $A[p]$, where $p = \lfloor(i-1)/d\rfloor + 1$. The time complexity is $O(\log_d n)$.

Algorithm SIFTUP-d
Input: An array $H[1..n]$ and an index i between 1 and n.
Output: $H[i]$ is moved up, if necessary, so that it is not larger than its parent.

```
1. done ← false
2. if $i = 1$ then exit      {node $i$ is the root}
3. repeat
4.     $p \leftarrow \lfloor(i-1)/d\rfloor + 1$
5.     if $key(H[i]) > key(H[p])$ then interchange $H[i]$ and $H[p]$
6.     else done ← true
7.     $i \leftarrow p$
8. until $i = 1$ or done
```

3.22. Rewrite Algorithm SIFT-DOWN for the case of d-heaps (see Exercise 3.21). What is its time complexity measured in terms of d and n?

The algorithm is shown in Procedure SIFTDOWN-d below. Note that in a d-ary heap A, the kth child of $A[i]$ is $A[q]$, where $q = di-(d-1)+k$. The **repeat** loop is iterated $\log_d n$ times, and in each iteration, the **for** loop is executed d times to find the maximum key. Hence, the time complexity is $O(d \log_d n)$.

Algorithm SIFTDOWN-d
Input: An array $H[1..n]$ and an index i between 1 and n.
Output: $H[i]$ is percolated down, if necessary, so that it is not smaller than its children.

```
1.  $done \leftarrow$ false
2.  if $di - (d-1) + 1 > n$ then exit      {node $i$ is a leaf}
3.  repeat
4.      $q \leftarrow di - (d-1)$
5.      $k \leftarrow 1$      {Set $q$ to first child}
6.      for $j \leftarrow 2$ to $d$
7.          if $q + j \leq n$ and $key(H[q+j]) > key(H[q+k])$ then $k \leftarrow j$
8.      end for
9.      $q \leftarrow q + k$
10.     if $key(H[i]) < key(H[q])$ then interchange $H[i]$ and $H[q]$
11.     else $done \leftarrow$ true
12.     end if
13.     $i \leftarrow q$
14. until $di - (d-1) + 1 > n$ or $done$
```

3.23. 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 $\{1, 2, \ldots, n\}$.

Assume for simplicity that n is a power of 2. Use the sequence of *unions* generated by Procedure UNION2 shown below, followed by any sequence of *finds*.

3.24. 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 $\{1, 2, \ldots, n\}$.

Algorithm 3.9 UNION2
```
1. if $n = 2$ then UNION$(1, 2)$ and exit
2. Recursively perform the union of $\{1, 2, \ldots, n/2\}$.
3. Recursively perform the union of $\{n/2 + 1, n/2 + 2, \ldots, n\}$.
4. UNION$(1, n/2 + 1)$
```

Use the sequence of *union*s generated by Procedure UNION2 shown in the previous exercise, followed by n copies of FIND(x), where x is any leaf in the resulting tree.

3.25. What are the ranks of nodes 3, 4 and 8 in Fig. 3.8(f)?

$rank(3) = 0$, $rank(4) = 2$, and $rank(8) = 3$.

3.26. 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.

The height of a node x either decreases or does not change after path compression, while its recorded rank does not change. This means that $rank(x)$ is an upper bound on the height of x.

3.27. Let σ 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.

The only operation which may take more than constant time is the *find* operation. Let u be a node in tree T with root x, and consider the instruction FIND(u). Let the path from u to x in T be $u = v_0, v_1, \ldots, v_k = x$ before path compression, where $k \geq 2$. So, using direct analysis, this *find* operation costs $O(k)$ time. However, using amortization, it is possible to show that it costs $O(1)$ amortized time. Charge two time units corresponding to v_{k-1} and v_k to the *find* instruction itself. Let w be any node on the path from u to x other than v_{k-1} and v_k. Let *makeset* be the operation of creating the first singleton set consisting of one node. Charge one time unit to the first *makeset*(w) instruction that resulted in a tree consisting of w only. Thus, the cost of the *find* is distributed among the *makeset* operations, and each *makeset* is assigned at most one time unit. After path compression, all nodes on the path from u to x will be directly connected to x, and the *makeset* operations for these nodes will not be charged again. Any subsequent find for these nodes will cost constant time.

3.28. 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.

The two heuristics are similar; the height of a tree after a sequence of *union*s is $O(\log n)$ with either heuristics. It is more natural to compare the heights of the two trees than to compare their sizes. However, it would be difficult to update the height of a tree after applying the path compression rule; the size of the tree remains unaltered.

3.29. Prove that the weight-balancing rule described in Exercise 3.28 guarantees that the resulting tree is of height $O(\log n)$.

We prove by induction on n that the height of the resulting tree is at most $\lfloor \log n \rfloor$, where n is the number of nodes in the resulting tree. For the base case, when $n = 1$, the height is $0 = \lfloor \log 1 \rfloor$. Now, suppose $n > 1$ and that any tree constructed by a sequence of *union* instructions and containing m nodes, for $m < n$, has height at most $\lfloor \log m \rfloor$. Consider a tree T with n nodes, and height h, that was constructed from two trees T_1 with n_1 nodes and height h_1, and T_2 with n_2 nodes and height h_2. Suppose that the root of T_2 was attached to the root of T_1. By the induction hypothesis, $h_1 \leq \lfloor \log n_1 \rfloor$, and $h_2 \leq \lfloor \log n_2 \rfloor$. The height of T is $\max\{h_1, h_2 + 1\}$. Clearly, $h_1 \leq \lfloor \log n \rfloor$. Since $n_2 \leq n/2$, $h_2 \leq \lfloor \log n \rfloor - 1$. It follows that $h \leq \lfloor \log n \rfloor$.

3.30. 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.

By Observation 3.1, for any node y, $rank(p(y)) \geq rank(y) + 1$. Moreover, before path compression, $p(y)$ was an ancestor of p, and hence the rank of $p(y)$ either increases or does not change after path compression.

3.31. 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.

The rank of the parent of v will change in subsequent *find* instructions and, in the worst case, it will assume all the values in $F(g-1), F(g-1)+1, F(g-1)+2, \ldots, F(g)-1$ before it acquires a higher group. The number of these ranks is $F(g) - F(g-1)$.

3.32. 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.

(a) No improvement needed; each *find* operation takes $O(1)$ time.

(b) Consider the sequence of $n-1$ *union*s:
$\textsc{union}(2,1)$, $\textsc{union}(3,2)$, $\textsc{union}(4,3)$,..., $\textsc{union}(n, n-1)$.
Executing these *union*s will cost $\Theta(n^2)$, assuming that the first element in L_1 is to be used as the name of the resulting set,

3.33. (Refer to Exercise 3.32). 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 $O(n \log n)$.

Let x be any element. Each time a list L containing x is merged with another list, the size of the new list containing x is at least doubled. Consider the number of times the pointer in item x to the set representative is changed. Since the size of the new set containing x is at least doubled after each merge, the number of these changes is $O(\log n)$. If we charge the change of representative to x itself, the total number of charges is $n \times O(\log n) = O(n \log n)$. Hence, the total cost is $O(n \log n)$.

4.11 Solutions

4.1. Give a recursive algorithm that computes the nth Fibonacci number f_n defined by

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

The recursive algorithm is shown as Algorithm FIBONACCI below.

```
Algorithm 4.13 FIBONACCI
Input: A positive integer $n$.
Output: $f_n$.
    1. $fib(n)$

Algorithm $fib(n)$.
    1. if $n \leq 2$ then return 1
    2. else return $fib(n-1) + fib(n-2)$
```

4.2. Give a recursive version of Algorithm SELECTIONSORT.

The recursive version is shown as Algorithm SELECTIONSORTREC below.

```
Algorithm 4.14 SELECTIONSORTREC
Input: An array $A[1..n]$ of $n$ elements.
Output: $A[1..n]$ sorted in nondecreasing order.
    1. $sort(1)$

Algorithm $sort(i)$      {Sort $A[i..n]$}
    1. if $i < n$ then
    2.     $k \leftarrow i$
    3.     for $j \leftarrow i+1$ to $n$
    4.         if $A[j] < A[k]$ then $k \leftarrow j$
    5.     end for
    6.     if $k \neq i$ then interchange $A[i]$ and $A[k]$
    7.     $sort(i+1)$
    8. end if
```

4.3. Give a recursive version of Algorithm INSERTIONSORT.

The recursive version is shown as Algorithm INSERTIONSORTREC below.

Algorithm 4.15 INSERTIONSORTREC
Input: An array $A[1..n]$ of n elements.

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

```
 1. $sort(n)$

Algorithm $sort(i)$      {Sort $A[1..i]$}
 1. if $i > 1$ then
 2.     $x \leftarrow A[i]$
 3.     $sort(i-1)$
 4.     $j \leftarrow i-1$
 5.     while $j > 0$ and $A[j] > x$
 6.         $A[j+1] \leftarrow A[j]$
 7.         $j \leftarrow j-1$
 8.     end while
 9.     $A[j+1] \leftarrow x$
10. end if
```

4.4. Give a recursive version of Algorithm BUBBLESORT given in Exercise 1.17.

The recursive version is shown as Algorithm BUBBLESORTREC below.

4.5. Derive a linear time and iterative version of Algorithm MAJORITY. Do not compute the median.

The algorithm is shown as Algorithm MAJORITYITERATIVE below.

4.6. Prove Observation 4.1.

Let k and k' be the number of occurrences of the candidate element before and after deleting two elements, respectively. Suppose that the candidate is a majority element before the deletion. Then, $k > n/2$. Note that $k' \geq k-1$. Hence,

$$k' \geq k - 1 > \frac{n}{2} - 1 = \frac{n-2}{2},$$

Algorithm 4.16 BUBBLESORTREC
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

```
1. sort(1)
```

Algorithm *sort*(i) {Sort $A[i..n]$}

```
1. if $i \leq n-1$ then
2.     for $j \leftarrow n$ downto $i+1$
3.         if $A[j] < A[j-1]$ then
4.             interchange $A[j]$ and $A[j-1]$
5.         end if
6.     end for
7.     $sort(i+1)$
8. end if
```

Algorithm 4.17 MAJORITYITERATIVE
Input: An array $A[1..n]$ of n elements.
Output: The majority element if it exists; otherwise *none*.

```
1. $j \leftarrow 1$;   $c \leftarrow A[1]$;   $count \leftarrow 1$
2. while $j < n$
3.     while $j < n$ and $count > 0$
4.         $j \leftarrow j+1$
5.         if $A[j] = c$ then $count \leftarrow count+1$
6.         else $count \leftarrow count-1$
7.     end while
8.     if $j < n$ then
9.         $j \leftarrow j+1$
10.        $c = A[j]$
11.        $count \leftarrow 1$
12.    end if
13. end while
14. $count \leftarrow 0$
15. for $j \leftarrow 1$ to $n$
16.     if $A[j] = c$ then $count \leftarrow count+1$
17. end for
18. if $count > \lfloor n/2 \rfloor$ then return $c$
19. else return none
```

that is, $k' > \frac{n-2}{2}$. It follows that the candidate is also a majority after the deletion of the two different elements.

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

False. Consider the case when n is even and the elements are distinct. Then, when $j = n$, $count = 0$ and there is no majority element.

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

False. Consider the case when n is odd and the elements are distinct. Then, when $j = n$, $count = 1 > 0$ and there is no majority element.

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

(a) $2^5 = 2(2^2)^2 = 2((2^1)^2)^2$. (b), (c) and (d) are similar.

4.10. Solve Exercise 4.9 using Algorithm EXP instead of Algorithm EXPREC.
(a) $n = 5 = 101$ in binary. Hence,
$1 \rightarrow 2 * 1 = 2 \rightarrow 2^2 = 4 \rightarrow 2 * 4^2 = 32$.
(b), (c) and (d) are similar.

4.11. Let A be a square matrix. Explain how to compute A^n efficiently, where n is a positive integer. How many matrix multiplications did you use?

Use the same algorithm as that for integer multiplication. This results in $\Theta(\log n)$ matrix multiplications.

4.12. Let A be a square matrix. Explain how to compute $A+A^2+\ldots+A^n$ efficiently, where n is a positive integer. Use the Θ-notation to express the number of matrix multiplications.

(a) Using integer exponentiation. Assume each power is evaluated separately.

(b) Using Horner's rule.

(a) Evaluate each power separately and add all powers. Number of multiplications is $\sum_{j=1}^{n} \Theta(\log j) = \Theta(n \log n)$.
(b) Using Horner's rule, compute $A(1 + A(1 + (\ldots A(1)\ldots)))$. There are $\Theta(n)$ matrix multiplications.

4.13. 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) $[1..2^n]$.

The number of iterations is $\Theta(\log m)$, where m is the range of values.
(a) $[1..n]$: $\Theta(n \log n)$.
(b) $[1..n^2]$: $\Theta(n \log n^2) = \Theta(n \log n)$.
(c) $[1..2^n]$: $\Theta(n \log 2^n) = \Theta(n^2)$.

4.14. 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).

The number of iterations in Algorithm RADIXSORT is $\Theta(\log n!) = \Theta(n \log n)$, and hence the running time of RADIXSORT becomes $\Theta(n^2 \log n)$. This is much more than the running time of Algorithm BOTTOMUPSORT, which is $\Theta(n \log n)$.

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

If the array sizes are changed dynamically starting from zero, then there should be no noticeable difference. However, if the array sizes are predetermined, the space complexity may be too high, as the size of each array may be as large as n.

4.16. 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.

Assume that the elements are uniformly distributed over the buckets so that each bucket has n/k elements. The initial phase of partitioning the items into k buckets takes $\Theta(n)$ operations in a reasonable implementation. Using Algorithm INSERTIONSORT, the sorting step takes $\Theta\left(k\left(\frac{n}{k}\right)^2\right)$ time. The final phase of combining buckets may require $\Theta(n)$ time in the worst case. Hence, the total running time is $\Theta\left(\frac{n^2}{k}\right)$. If, for example, k is set to $n/10$, the algorithm will run in linear time. However, if the input is not uniformly distributed, then all numbers may fall into one bucket in the worst case, which may result in a running time of $\Theta(n^2)$.

4.17. Instead of using another sorting algorithm in Exercies 4.16, design a recursive version of bucket sort that recursively sorts the numbers in each bucket. What is the major disadvantage of this recursive version?

Instead of using another sorting algorithm, *bucket sort* would simply call itself recursively on each bucket. However, there is a lot of bookkeeping involved due to the huge number of recursive calls to create smaller and smaller buckets. This makes recursion impractical in *bucket sort*.

4.18. 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.

(a) SELECTIONSORT: Stable.
(b) INSERTIONSORT: Stable.
(c) BUBBLESORT: Stable.
(d) BOTTOMUPSORT: Stable. Consider the number 1 that occurs on the left and right, that is, 1_1 to the left and 1_2 to the right. Then, Algorithm MERGE will put them in the correct order

as $1_1, 1_2$.

(e) HEAPSORT is not stable. To see this, consider sorting the input $1_1, 1_2$. Then, 1_1 and 1_2 will be interchanged by the algorithm.

(f) RADIXSORT: Stable. Elements will be inserted into lists in correct order.

4.19. Let $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ be a polynomial of degree $n-1$, where n is a power of 2. Design a recursive algorithm to implement Horner's rule to evaluate $f(x)$ at the point $x = b$. What is the time complexity of your algorithm?

The algorithm is shown below as Algorithm POLYNOMIAL. It recursively computes $y = a_1 + a_2x + a_3x^2 + \ldots + a_{n-1}x^{n-2}$, and returns $a_0 + yx$ evaluated at the point $x = b$. Its time complexity is $\Theta(n)$.

Algorithm 4.18 POLYNOMIAL
Input: An array $A[1..n]$ of n elements corresponding to $a_0, a_1, \ldots, a_{n-1}$ and x.
Output: $a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$

```
    1. $p = poly(A, b, 1)$
    2. return $p$

Algorithm  $poly(A, x, low)$
    1. if $low = n$ then return $A[low]$
    2. else
    3.     $y \leftarrow poly[A, x, low + 1]$
    4.     $w \leftarrow A[low] + xy$
    5.     return $w$
    6. end if
```

4.20. Modify Algorithm SUBSETS1 in Sec. 4.6 for generating the subsets of numbers so that it outputs only m-subsets, that is, subsets of size m.

The algorithm is shown as Algorithm SUBSETS4 below. Note that the statement $size = size + j$ acts like the conditional statement **if** $j = 1$ **then** $size = size + 1$.

4.21. Modify Algorithm SUBSETS1 in Sec. 4.6 for generating the subsets of numbers so that it outputs all subsets containing 1 before all

Algorithm 4.19 SUBSETS4
Input: Two positive integers n and m.
Output: Subsets of the numbers $1, 2, \ldots, n$ of size m.

```
 1. for $j \leftarrow 1$ to $n$
 2.     $v[j] \leftarrow 0$
 3. end for
 4. $sub4(1)$
```

Algorithm *sub4*(k)

```
 1. if $k \leq n$ then
 2.     if $k = 1$ then $size = 0$
 3.     for $j \leftarrow 0$ to 1
 4.         $v[k] \leftarrow j$
 5.         $size = size + j$
 6.         $sub4(k + 1)$
 7.         if $k = n$ and $size = m$ then
 8.             for $i \leftarrow 1$ to $n$ if $v[i] = 1$ then output $i$
 9.         if $v[k] = 1$ then $v[k] = 0$
10.         $size = size - j$
11.     end for
12. end if
```

subsets without 1.

Change the **for** statement
"**for** $j \leftarrow 0$ **to** 1" to "**for** $j \leftarrow 1$ **downto** 0".
In this case, the output for $n = 3$ would be
$\{1,2,3\}, \{1,2\}, \{1,3\}, \{1\}, \{2,3\}, \{2\}, \{3\}, \{\}$.

4.22. Do Exercise 4.20 for Algorithm SUBSETS2.

Change the statement
if $k = n$ **then for** $j \leftarrow 1$ **to** i **output** $A[j]$
to
if $k = n$ **and** $i = m$ **then for** $j \leftarrow 1$ **to** i **output** $A[j]$,
and include m in the input.

4.23. Do Exercise 4.21 for Algorithm SUBSETS2.

The modified algorithm is shown below as Algorithm SUBSETS5. In this case, the output for $n = 3$ is
$\{1,2,3\}, \{1,2\}, \{1,3\}, \{1\}, \{2,3\}, \{2\}, \{3\}, \{\}$.

```
Algorithm 4.20 SUBSETS5
Input: A positive integer $n$.
Output: All subsets of the numbers $1, 2, \ldots, n$.
    1. $i \leftarrow 0$
    2. $sub5(1)$

Algorithm $sub5(k)$
    1. if $k \leq n$ then
    2.     $i \leftarrow i + 1$
    3.     $A[i] \leftarrow k$
    4.     $sub5(k + 1)$
    5.     if $k = n$ then for $j \leftarrow 1$ to $i$ output $A[j]$
    6.     $i \leftarrow i - 1$
    7.     $sub5(k + 1)$
    8.     if $k = n$ then for $j \leftarrow 1$ to $i$ output $A[j]$
    9. end if
```

4.24. Explain why in Algorithm SUBSETS2 in Sec. 4.6, the statement $i \leftarrow i - 1$ is necessary.

If this statement is deleted, $A[i]$ will be prepended to all subsequent subsets. For example, when $n = 2$, the output will be

$$\{\}, \{2\}, \{2, 1\}, \{2, 1, 2\} \text{ instead of } \{\}, \{2\}, \{1\}, \{1, 2\}.$$

4.25. 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.

Before the recursive call, they are interchanged so that $P[m..n] = j, m + 1, \ldots, n$. Therefore, they must be interchanged back after the recursive call so that $P[m..n] = m, m + 1, \ldots, n$. This is done so that the permutations of the elements $\{m, m + 1, \ldots, n\}$ are generated correctly. For example, if they are not interchanged back, the output when $n = 3$ will be

$$\{1, 2, 3\}, \{1, 3, 2\}, \{3, 1, 2\}, \{3, 2, 1\}, \{1, 2, 3\}, \{1, 3, 2\}.$$

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

If it is not reset to 0, only one permutation will be printed, since there are no free positions. For example, if it is not reset to 0 and $n = 4$, the output will consist of the permutation $\{4, 3, 2, 1\}$ only.

4.27. 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.

Before Procedure *perm2* is invoked with $m-1$, the number of zeros is reduced by 1. After this call is completed, the number of zeros is increased by 1. Since initially the number of zeros is n, the number of zeros during the call *perm2*(m) is exactly m.

4.28. 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.

Change the **for** loop statement to: **for** $j \leftarrow n$ **downto** 1.

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

The algorithm is shown below as Algorithm SUMREC.

```
Algorithm 4.21 SUMREC
Input: An array $A[1..n]$ of $n$ integers, and an integer $x$.
Output: Two integers in $A$ whose sum is $x$.
    1. $sum(A, x)$

Algorithm $sum(A, x)$
    1. if $|A| = 2$, then if $A[1] + A[2] = x$ then return $(A[1], A[2])$
       else return $(\infty, \infty)$
    2. $y = A[1] + A[n]$
    3. if $y = x$ then return $(A[1], A[n])$
    4. else if $y > x$ then return $sum(A[1..n-1], x)$
    5. else return $sum(A[2..n], x)$
```

4.30. Convert the algorithm in Exercise 4.29 into an iterative algorithm.

The algorithm is shown below as Algorithm SUMITERATIVE.

Algorithm 4.22 SUMITERATIVE
Input: An array $A[1..n]$ of n integers, and an integer x.
Output: Two integers in A whose sum is x.

```
1. $low = 1$;    $high = n$
2. while $low < high$
3.     if $high - low = 1$, then if $A[low] + A[high] = x$
       then return $(A[low], A[high])$
       else return $(\infty, \infty)$
4.     $y = A[low] + A[high]$
5.     if $y = x$ then return $(A[low], A[high])$
6.     else if $y > x$ then $high \leftarrow high - 1$
7.     else $low \leftarrow low + 1$
8. end while
```

4.31. Let a and b be two positive integers. The greatest common divisor of a and b, denoted by $\gcd(a, b)$, is the largest integer that divides both a and b. Derive a recursive algorithm to compute $\gcd(a, b)$. Hint: If c divides a and b, then c divides their difference.

The recursive algorithm is shown as Algorithm EUCLIDREC below. See also Exercise 1.41.

Algorithm 4.23 EUCLIDREC
Input: Two positive integers a and b.
Output: $\gcd(a, b)$.

```
1. $\gcd(a, b)$
```

Algorithm $\gcd(a, b)$.

```
1. $r \leftarrow a \bmod b$
2. if $r = 0$ return $b$
3. else return $\gcd(b, r)$
```

4.32. Derive the running time of the algorithm in Exercise 4.31. Note that the running time is in terms of the size of the input, which is the number of digits in $\max\{a, b\}$.

See Exercise 1.41.

4.33. Convert the algorithm in Exercise 4.31 into an iterative algorithm.

An efficient iterative algorithm for computing the gcd of two numbers was given in Exercise 1.41.

See Exercise 1.41.

4.34. Use induction to solve Exercise 2.7.

Let stack S_1 contain the input, and let S_2 be a temporary stack that is initially empty. Suppose we move the minimum element from S_1 to S_2, and recursively sort the rest of the elements in the input stack. By induction, the algorithm will sort the rest of the elements and put them on top of the minimum in the temporary stack S_2. Therefore, the algorithm will sort all elements in the input stack.
This implies the following algorithm to sort the numbers. Prepare an empty stack S_2. While input stack S_1 is not empty do the following: (1) Pop an element x from S_1. (2) While stack S_2 is not empty and its top is greater than x, pop from S_2 and push into S_1. (3) Push x into S_2. At the end, the sorted numbers are in stack S_2. The details are shown in Algorithm STACKSORTING below. After x is set to the minimum, all elements in S_2 are popped and pushed back into S_1, and x is pushed into S_2. At this point, S_2 contains the minimum, and S_1 contains the rest, and the remaining elements are sorted recursively.
The running time is $O(n^2)$, as there are $O(n^2)$ stack operations, as can be verified by considering an input that is sorted in reverse order.

4.35. Use induction to solve Exercise 2.8.

Assume for simplicity that all elements are distinct. (1) Move all items from stack S_1 to stack S_2, store maximum value found in x. (2) Move all items from stack S_2 to stack S_1, except x, which is moved to S_3 instead. (3) Repeat until both S_1 and S_2 are empty. The time complexity is $\Theta(n^2)$.

Algorithm 4.24 STACKSORTING
Input: An array $A[1..n]$ of n numbers in stack S_1.

Output: A sorted in ascending order.

```
 1. Initialize $S_2 = \{\}$.
 2. **while** $S_1 \neq \{\}$
 3.     $x \leftarrow pop(S_1)$
 4.     **while** top of stack $S_2 \geq x$ **and** $S_2 \neq \{\}$
 5.         $y \leftarrow pop(S_2)$
 6.         $push(S_1, y)$
 7.     **end while**
 8.     $push(S_2, x)$
 9. **end while**
10. **return** $S_2$
```

5.16 Solutions

5.1. Give a divide-and-conquer algorithm that returns a pair (x, y) where x is the largest number and y is the the second largest number in an array of n numbers. Derive the time complexity of your algorithm.

Algorithm SECOND1 below finds the largest and second largest. Assuming n is a power of 2, the number of comparisons is governed by the following recurrence relation.

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

This recurrence solves to $C(n) = (3n/2) - 2$.

Algorithm 5.11 SECOND1
Input: An array $A[1..n]$ of n integers, where n is a power of 2.
Output: (x, y):the largest and second largest in A.

```
 1. $(x, y) \leftarrow second(1, n)$
 2. return $(x, y)$
```

Algorithm *second*(*low*, *high*)

```
 1. if $high - low = 1$ then
 2.     $x \leftarrow \max\{A[low], A[high]\}$
 3.     $y \leftarrow \min\{A[low], A[high]\}$
 4.     return $(x, y)$
 5. else
 6.     $mid \leftarrow \lfloor (low + high)/2 \rfloor$
 7.     $(x_1, y_1) \leftarrow second(low, mid)$
 8.     $(x_2, y_2) \leftarrow second(mid + 1, high)$
 9.     if $x_1 \geq x_2$ then
10.         return $(x_1, \max\{x_2, y_1\})$
11.     else
12.         return $(x_2, \max\{x_1, y_2\})$
13.     end if
14. end if
```

5.2. Repeat Execise 5.1 for finding the largest and second largest elements using $n + \log n - 2$ comparisons. Hint: See Sec. 11.3.4.2.

Algorithm SECOND2 below first finds the maximum x and a list R

of candidates for second largest. The list R is computed as follows. Let (x, P) be the maximum and list of candidates in the left half, and (y, Q) be the maximum and list of candidates in the right half. If $x \geq y$, then Q is discarded, and y is appended to P, which is returned as the set of candidates. On the other hand, if $x < y$, then P is discarded, and x is appended to Q, which is returned as the set of candidates.
Assuming n is a power of 2, the number of comparisons for finding the pair (x, R) is governed by the following recurrence relation.

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

This recurrence solves to $C(n) = n - 1$. Finding the second maximum in R takes $|R| = \log n - 1$ additional comparisons for a total of $n - 1 + \log n - 1 = n + \log n - 2$ comparisons.

Algorithm 5.12 SECOND2
Input: An array $A[1..n]$ of n integers, where n is a power of 2.

Output: (x, y):the largest and second largest in A.

```
 1. $(x, R) \leftarrow$ $max(1, n)$
 2. $y \leftarrow \max R$
 3. **return** $(x, y)$
```

Algorithm *max*$(low, high)$

```
 1. **if** $high - low = 1$ **then**
 2.     $x \leftarrow \max\{A[low], A[high]\}$
 3.     $y \leftarrow \min\{A[low], A[high]\}$
 4.     **return** $(x, \{y\})$
 5. **else**
 6.     $mid \leftarrow \lfloor (low + high)/2 \rfloor$
 7.     $(x, P) \leftarrow$ $max(low, mid)$
 8.     $(y, Q) \leftarrow$ $max(mid + 1, high)$
 9.     **if** $x \geq y$ **then**
10.         Append $y$ to $P$
11.         **return** $(x, P)$
12.     **else**
13.         Append $x$ to $Q$
14.         **return** $(y, Q)$
15.     **end if**
16. **end if**
```

5.3. 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.

The modified algorithm is shown below as Algorithm MINMAX2. Its number of comparisons is given by the recurrence:

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2 & \text{if } n > 2. \end{cases}$$

The solution to this recurrence is $C(n) = 3n/2 - 2$ if n is a power of 2. Thus, more than $\lfloor 3n/2 - 2 \rfloor$ comparisons may be needed if n is not a power of 2. For example, $C(3) = 3, C(6) = 8$, etc. (See Exercise 5.4 for the case when n is not necessarily a power of 2).

Algorithm 5.13 MINMAX2
Input: An array $A[1..n]$ of n integers.
Output: (x, y):the minimum and maximum integers in A.

```
1. $minmax(1, n)$

Algorithm $minmax(low, high)$
 1. if $high = low$ then return $(A[low], A[low])$
 2. else if $high - low = 1$ then
 3.     if $A[low] < A[high]$ then return $(A[low], A[high])$
 4.     else return $(A[high], A[low])$
 5.     end if
 6. else
 7.     $mid \leftarrow \lfloor (low + high)/2 \rfloor$
 8.     $(x_1, y_1) \leftarrow minmax(low, mid)$
 9.     $(x_2, y_2) \leftarrow minmax(mid + 1, high)$
10.     $x \leftarrow \min\{x_1, x_2\}$
11.     $y \leftarrow \max\{y_1, y_2\}$
12.     return $(x, y)$
13. end if
```

5.4. 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.

Let $x = \min\{A[1], \ldots, A[n-2]\}$ and $y = \max\{A[1], \ldots, A[n-2]\}$. To compute the minimum and maximum in A, first compare $A[n-$

1] with $A[n]$. Let the minimum of these two numbers be p, and the maximum be q. Update x to be the minimum of x and p, and update y to be the maximum of y and q. This leads to Algorithm MINMAX3 shown below. There are three element comparisons in each iteration of the **while** loop plus one comparison before the loop. The total is $3(n-2)/2+1 = 3n/2-2$ comparisons for even n (even if it is not a power of 2).

Algorithm 5.14 MINMAX3
Input: An array $A[1..n]$ of n integers, where n is a power of 2.

Output: (x, y):the minimum and maximum integers in A.

```
 1. $x \leftarrow \min\{A[1], A[2]\}$
 2. $y \leftarrow \max\{A[1], A[2]\}$
 3. $j \leftarrow 2$
 4. **while** $j < n$
 5.     $p \leftarrow \min\{A[j+1], A[j+2]\}$
 6.     $q \leftarrow \max\{A[j+1], A[j+2]\}$
 7.     $x \leftarrow \min\{x, p\}$
 8.     $y \leftarrow \max\{y, q\}$
 9.     $j \leftarrow j+2$
10. **end while**
```

5.5. 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.

Let $x = \min\{x_1, x_2\}$ and $y = \max\{x_1, x_2\}$. First, search for x. Let k be the result of searching for x, that is, $x = A[k]$ if x is in A, or 0 otherwise. Next, search for y in $A[k+1..n]$. The running time is still $O(\log n)$.

5.6. 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.

The algorithm is shown as Algorithm BINARYSEARCH2 below.
Each iteration reduces the search range by a factor of at least 3/2.
Hence, the running time is $O(\log_{3/2} n) = O(\log n)$.

```
Algorithm 5.15 BINARYSEARCH2
Input: An array $A[1..n]$ of $n$ elements sorted in nondecreasing order and
       an element $x$.
Output: $j$ if $x = A[j], 1 \leq j \leq n$, and 0 otherwise.
    1. $binarysearch(1, n)$

Algorithm binarysearch(low, high)
    1. if $low > high$ then return 0
    2. else
    3.     $d \leftarrow \lfloor (high - low)/3 \rfloor$
    4.     $mid \leftarrow low + d$
    5.     if $x = A[mid]$ then return $mid$
    6.     else if $x < A[mid]$ then return $binarysearch(low, mid - 1)$
    7.     else return $binarysearch(mid + 1, high)$
    8. end if
```

5.7. 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.

The algorithm is shown as Algorithm BINARYSEARCH3 below.
Each iteration reduces the search range by a factor of at least 3. Hence, the running time is $O(\log_3 n) = O(\log n)$.

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

Proof. By induction on n, the size of the array.
Basis step: If $n = 1$, then there is only one element, which is sorted.
Induction step: Suppose the hypothesis holds for all arrays with less than n elements. We show that it also holds for n. By induction, the two halves will be sorted, and by correctness of Algorithm MERGE, the whole array will be sorted. Thus, the hypothesis holds for n, and hence the algorithm sorts its input correctly. □

5.9. It was shown in Sec. 5.3 that algorithms BOTTOMUPSORT and MERGESORT are very similar. Give an example of an array of numbers in which

Algorithm 5.16 BINARYSEARCH3
Input: An array $A[1..n]$ of n elements sorted in nondecreasing order and an element x.
Output: j if $x = A[j], 1 \leq j \leq n$, and 0 otherwise.

```
 1. binarysearch(1, n)
```

Algorithm *binarysearch*(*low*, *high*)

```
 1. if $low > high$ then return 0
 2. else
 3.     $d \leftarrow \lfloor (high - low)/3 \rfloor$
 4.     $mid1 \leftarrow low + d$
 5.     $mid2 \leftarrow low + 2d$
 6.     if $x = A[mid1]$ then return $mid1$
 7.     else if $x = A[mid2]$ then return $mid2$
 8.     else if $x < A[mid1]$ then return binarysearch($low, mid1 - 1$)
 9.     else if $x > A[mid2]$ then return binarysearch($mid2 + 1, high$))
10.     else return binarysearch($mid1 + 1, mid2 - 1$)
11. end if
```

(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.

(a) Consider any input of n elements, where n is a power of 2.

(b) Consider input $\boxed{1 \mid 2 \mid 3}$. Algorithm BOTTOMUPSORT first compares 1 and 2, then it merges 1, 2 with 3 using two comparisons for a total of three comparisons.
On the other hand, Algorithm MERGESORT first compares 2 and 3, then it merges 1 with 2, 3 using one comparison for a total of two comparisons.

(c) Consider input $\boxed{3 \mid 2 \mid 1}$. Algorithm BOTTOMUPSORT first compares 3 and 2, then it merges 2, 3 with 1 using one comparison for a total of two comparisons.
On the other hand, Algorithm MERGESORT first compares 2 and 1, then it merges 3 with 1, 2 using two comparisons for a total of three comparisons.

5.10. 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.

(a) The modified algorithm is shown below as Algorithm MERGESORT2.

```
Algorithm 5.17 MERGESORT2
Input: An array $A[1..n]$ of $n$ elements.
Output: $A[1..n]$ sorted in nondecreasing order.
    1. $mergesort(A, 1, n)$

Algorithm $mergesort(A, low, high)$
    1. if $low < high$ then
    2.     $d \leftarrow \lfloor (high - low)/4 \rfloor$.
    3.     $mergesort(A, low, low + d)$
    4.     $mergesort(A, low + d + 1, low + 2d)$
    5.     $mergesort(A, low + 2d + 1, low + 3d)$
    6.     $mergesort(A, low + 3d + 1, high)$
    7.     MERGE $(A, low, low + d, low + 2d)$
    8.     MERGE $(A, low + 2d + 1, low + 3d, high)$
    9.     MERGE $(A, low, low + 2d, high)$
   10. end if
```

(b) The running time of the algorithm is given by the recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 4T(n/4) + bn & \text{if } n \geq 2, \end{cases}$$

whose solution is $T(n) = \Theta(n \log_4 n) = \Theta(n \log n)$.

5.11. What will be the running time of the modified algorithm in Exercise 5.10 if the input array is divided into k parts instead of 4? Here, k is a *fixed* positive integer greater than 1.

The running time of the algorithm is given by the recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ kT(n/k) + bn & \text{if } n \geq 2, \end{cases}$$

whose solution is $T(n) = \Theta(n \log_k n) = \Theta(n \log n)$ since k is fixed.

5.12. 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 *high* $-$ *low* $+ 1 \leq m$ **then** INSERTIONSORT(A[*low*..*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.

The maximum number of comparisons done by the algorithm is given by the recurrence

$$C(n) = \begin{cases} m(m-1)/2 & \text{if } n = m \\ 2C(n/2) + n - 1 & \text{if } n > m \end{cases}$$

If we expand this recurrence i times, we obtain

$$C(n) = 2^i \times C(n/2^i) + in - (2^i - 1) \leq 2^i \times C(n/2^i) + kn,$$

where $k = \log n$. Letting $m = n/2^i$ yields

$$C(n) \leq \frac{n}{m}\frac{m(m-1)}{2} + kn \leq cnm + n \log n,$$

for some constant c. It follows that $m \leq b \log n$ for some constant b.

5.13. Use Algorithm SELECT to find the kth smallest element in the list of numbers given in Example 5.1, where
(a) $k = 1$. (b) $k = 9$. (c) $k = 17$. (d) $k = 22$. (e) $k = 25$.

Similar to Example 5.1.

5.14. 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.

The algorithm will go into infinite loop.

5.15. 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 Algorithm *select* in Algorithm SELECT will there be? Explain your answer clearly.

Assume without loss of generality that A consists of the numbers $1, 2, \ldots, 105$. In the first iteration, there are $10 \times 5 + 3 = 53$ elements less than or equal to the median of medians mm. It is at index 53, which means that mm coincides with the true median. In the second iteration, the number of remaining elements is 53, and there are $4 \times 5 + 3 = 23$ elements less than or equal to the median of medians. After the second iteration, the number of remaining elements is ≤ 44, and hence they will be sorted and the 17th smallest element will be returned.

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

Assume without loss of generality that A consists of the numbers $1, 2, \ldots, n$. There are two cases. First, if $\lfloor n/5 \rfloor$ is odd. In this case, the number of remaining elements is $5\lfloor \lfloor n/5 \rfloor /2 \rfloor + 3$. Secondly, if $\lfloor n/5 \rfloor$ is even, then the number of remaining elements is $5\lfloor \lfloor n/5 \rfloor /2 - 1 \rfloor + 3$. In both cases, he number of remaining elements is approximately $n/2$. This is similar to Algorithm BINARYSEARCHREC.

5.17. In Algorithm SELECT, groups of size 5 are sorted in each invocation of the algorithm. This means that finding an algorithm 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.

Assume that the numbers are a, b, c, d, e. Compare a to b and c to d. Suppose without loss of generality that $a < b$ and $c < d$.

Compare a to c. Suppose without loss of generality that $a < c$. Insert e into $\langle a, c, d\rangle$. This can be done with two comparisons. Insert b into $\{e, c, d\}$. This can be done with two comparisons. The total number of comparisons is seven.

5.18. 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 5.17).

We count the number of comparisons $C(n)$ as follows. Step 4 takes at most $7\lfloor n/5\rfloor \leq 7n/5$ comparisons. Step 6 takes n comparisons. Hence, substituting in the recurrence

$$C(n) \leq \begin{cases} c & \text{if } n < 44 \\ C(\lfloor n/5\rfloor) + C(\lfloor 3n/4\rfloor) + cn & \text{if } n \geq 44, \end{cases}$$

for $c = 12/5$ yields the inequality $C(n) \leq \frac{cn}{1-1/5-3/4} = 20cn$, where $c = 12/5$, that is, $C(n) \leq 48n$.
It can be shown using more detailed analysis of the algorithm that $C(n) \leq 16n$. Empirical results show that $C(n) \leq 10n$.

5.19. Based on the number of comparisons counted in Exercise 5.18, determine for what values of n one should use a straightforward sorting method and extract the kth element directly.

We will assume that the number of comparisons is $C(n) \leq 10n$. Since sorting costs at most $n \log n$ comparisons, setting $n \log n \leq 10n$ gives $\log n \leq 10$ or $n \leq 2^{10} = 1024$. Thus, sorting should be used for roughly $n \leq 1024$.

5.20. 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?

We count the number of comparisons $C(n)$ as follows. Step 4 costs at most $(n/g)g \log g = n \log g$ comparisons. Step 6 takes n comparisons. Assume $g \geq 5$, since for $g = 3, 4$, the running time is $\Theta(n \log n)$ (see Exercise 5.21). Hence, the recurrence becomes

$$C(n) \leq \begin{cases} c & \text{if } n < 44 \\ C(\lfloor n/5 \rfloor) + C(\lfloor 3n/4 \rfloor) + (\log g + 1)n & \text{if } n \geq 44, \end{cases}$$

Substituting in the inequality $C(n) \leq \frac{cn}{1-1/5-3/4} = 20cn$ for $c = \log g + 1$, yields $C(n) \leq 20(\log g + 1)n$. Thus, when g is too large, the multiplicative constant explodes.

5.21. 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 5.20).

Only 5, 7, 9, 11 guarantees $\Theta(n)$ worst case performance. To see this, note that the recurrence for the running time is

$$T(n) \leq \begin{cases} c & \text{if } n < d \\ T(\lfloor n/g \rfloor) + T(\lfloor 3n/4 \rfloor) + cn & \text{if } n \geq d, \end{cases}$$

for some constant d, where g is the group size. It follows by Theorem 1.5, that the solution to this recurrence is $T(n) = \Theta(n)$ if and only if $1/g + 3/4 < 1$ or $g > 4$.

5.22. 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?

The modified algorithm is shown as Algorithm SELECT2 below. An advantage of the modified algorithm is that it uses less space.

5.23. 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?

Assume for simplicity that n is a power of 2. Let $a = A[n/2]$ and $b = B[n/2]$. If $a \leq b$, then any element in $A[1..n/2]$ is less than $n/2$ elements in A and $n/2$ elements in B and cannot be the median, so all elements in $A[1..n/2]$ are discarded. Also, any element in $B[n/2 + 1..n]$ is greater than $n/2$ elements in B and $n/2$ elements in A and cannot be the median, so all elements in $B[n/2+1..n]$ are

```
Algorithm 5.18 SELECT2
Input: An array $A[1..n]$ of $n$ elements and an integer $k$, $1 \le k \le n$.
Output: The $k$th smallest element in $A$.

    1. $select(A, low, high, k)$

Algorithm select(A, low, high, k)
    1. $n \leftarrow high - low + 1$
    2. if $n < 44$ then sort $A$ and return $A[k]$
    3. Let $q = \lfloor n/5 \rfloor$. Divide $A$ into $q$ groups of 5 elements each. If 5 does
       not divide $p$, then discard the remaining elements.
    4. Sort each of the $q$ groups individually and extract its median. Let
       the set of medians be $M$.
    5. $mm \leftarrow select(M, 1, q, \lceil q/2 \rceil)$    {$mm$ is the median of medians}
    6. Interchange $mm$ with $A[low]$
    7. SPLIT($A[low..high]$, $w$)   {$w$ is the new position of $mm$}
    8. case
           $k = w$: return $mm$
           $k < w$: $high \leftarrow w - 1$
           $k > w$: $low \leftarrow w + 1$;  k ← k - w
    9. end case
   10. s$select(A, low, high, k)$
```

discarded. The case $b \le a$ is symmetrical. The algorithm is shown as Algorithm MEDIAN below. Its time complexity is $\Theta(\log n)$.

```
Algorithm 5.19 MEDIAN
Input: Two arrays $A[1..n]$ and $B[1..n]$ of $n$ elements each, where $n$ is a power
       of 2.
Output: The median element in $A \cup B$.

    1. $median(A, B, n)$

Algorithm median(A, B, n)
    1. if $n = 1$ then return $\min\{A[1], B[1]\}$ and exit
    2. $a = A[n/2]$;   $b = B[n/2]$
    3. if $a \le b$ then
    4.     Discard $A[1..n/2]$ and $B[n/2 + 1..n]$
    5. else
    6.     Discard $B[1..n/2]$ and $A[n/2 + 1..n]$
    7. end if
    8. $median(A, B, n/2)$
```

5.24. Make use of the algorithm obtained in Exercise 5.23 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).

5.25. 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.

First, find the kth smallest element, call it x. Next, scan the array A and return all elements less than or equal to x.

5.26. 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?

(a) $f(n) = 0$ if $A[low]$ is the minimum or the maximum, e.g. $\boxed{1}\boxed{2}\boxed{3}\boxed{4}$ and $\boxed{4}\boxed{1}\boxed{2}\boxed{3}$.

(b) The maximum value of $f(n)$ is $n-2$. It is achieved when $A[low]$ is the second maximum and $A[low + 1]$ is the maximum, e.g. $\boxed{3}\boxed{4}\boxed{1}\boxed{2}$.

5.27. 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.

Find the median and exchange it with $A[low]$ before Algorithm SPLIT starts. This will improve the running time of Algorithm QUICKSORT substantially.

5.28. 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?

See Algorithm PARTITION below. The number of comparisons is $n - 1$.

Algorithm 5.20 PARTITION
Input: An array of elements $A[low..high]$.
Output: (1)A with its elements rearranged, if necessary, as described above.
(2) w, the new position of the splitting element A$[low]$.

```
 1. $i \leftarrow low + 1$;   $j \leftarrow high$;   $x \leftarrow A[low]$
 2. while $i < j$
 3.     while $A[i] \leq x$
 4.         $i \leftarrow i + 1$
 5.     end while
 6.     while $A[j] > x$
 7.         $j \leftarrow j - 1$
 8.     end while
 9.     if $i < j$ then interchange $A[i]$ and $A[j]$
10. end while
11. $A[low] = A[j]$;   $A[j] = x$
12. $w \leftarrow j$
13. return $A$ and $w$
```

5.29. Let $A[1..n]$ be a sequence 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 $\Theta(n)$.

See Algorithm PARTITION2 below.

5.30. Convert Algorithm QUICKSELECT into an iterative algorithm.

The iterative version is shown as Algorithm QUICKSELECTITERATIVE below. Algorithm SPLIT is the partitioning algorithm presented in Sec. 5.6.1.

Algorithm 5.21 PARTITION2
Input: An array of elements $A[low..high]$.
Output: A with its elements rearranged, if necessary, as described above.

```
1. $i \leftarrow low$;   $j \leftarrow high$
2. while $i < j$
3.     while $A[i] \le 0$
4.         $i \leftarrow i + 1$
5.     end while
6.     while $A[j] > 0$
7.         $j \leftarrow j - 1$
8.     end while
9.     if $i < j$ then interchange $A[i]$ and $A[j]$
10. end while
11. return $A$
```

Algorithm 5.22 QUICKSELECTITERATIVE
Input: An array $A[1..n]$ of n elements and an integer k, $1 \le k \le n$.
Output: The kth smallest element in A.

```
1. $low = 1$;   $high = n$
2. while $low \le high$
3.     SPLIT($A[low..high]$, $w$)   {$w$ is the new position of $A[low]$}
4.     case
           $k = w$ : return $A[w]$
           $k < w$ : $high = w - 1$
           $k > w$ : $low = w + 1$
5.     end case
6. end while
```

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

The space complexity is $\Theta(1)$, except for the space needed for the stack (see Exrcise 5.33). In the best case, it is given by the recurrence $S(n) = S(n/2) + c$, or $S(n) = \Theta(\log n)$. In the worst case, it is given by the recurrence $S(n) = S(n-1) + c$, or $S(n) = \Theta(n)$. The average space complexity is given by the recurrence $S(n) = S(n/2) + c$, or $S(n) = \Theta(\log n)$.

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

The behavior of Algorithm QUICKSORT in this case is that of sorted input, that is, the running time is $\Theta(n^2)$.

5.33. Give an iterative version of Algorithm QUICKSORT.

The iterative version is shown as Algorithm QUICKSORTITERATIVE below. In the algorithm, *St* is the stack, and *push* and *pop* are the push and pop operations on the stack. Algorithm SPLIT is the partitioning algorithm presented in Sec. 5.6.1.

Algorithm 5.23 QUICKSORTITERATIVE
Input: An array $A[1..n]$ of n elements.

Output: The elements in A sorted in nondecreasing order.

```
 1. $low = 1$;    $high = n$
 2. push $(1, n)$
 3. while $St \neq \{\}$    {$St$ is the stack}
 4.     pop $(low, high)$
 5.     if $low < high$ then
 6.         SPLIT($A[low..high]$, $w$)   {$w$ is the new position of $A[low]$}
 7.         If $high > w + 1$ then push $(w + 1, high)$
 8.         If $w - 1 > low$ then push $(low, w - 1)$
 9.     end if
10. return $A$
```

5.34. Which of the following sorting algorithms are stable (see Exercise 4.18)?
(a)HEAPSORT (b)MERGESORT (c)QUICKSORT.

We will use subscripts to denote the order of equal numbers.

(a) HEAPSORT is not stable. To see this, consider sorting the input $1_1, 1_2$. Then, 1_1 and 1_2 will be interchanged by the algorithm.

(b) MERGESORT is stable. Consider the number 1 that occurs on the left and right, that is, 1_1 to the left and 1_2 to the right. Then, Algorithm MERGE will put them in the correct order as $1_1, 1_2$.

(c) QUICKSORT is not stable. To see this, consider sorting the input $1_1, 1_2$. Then, 1_1 and 1_2 will be interchanged by Algorithm SPLIT.

5.35. 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.

INSERTIONSORT, HEAPSORT, BOTTOMUPSORT, MERGESORT and QUICKSORT are adaptive. The others are not adaptive.

5.36. 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.

Compute $(ad + bc)$ from $ad + bc = (a + b)(c + d) - ac - bd$.

5.37. 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.

There are some approaches. The easiest approach is to add extra rows and columns of 0's to make dimensions powers of 2.

5.38. Let $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ be a polynomial of degree $n - 1$, where n is a power of 2. Design a divide and conquer algorithm to implement Horner's rule to evaluate $f(x)$ at the point $x = b$. What is the time complexity of your algorithm?

The algorithm is shown below as Algorithm POLYNOMIAL. It recursively computes $y = a_0 + a_1x + a_2x^2 + \ldots + a_{n/2-1}x^{n/2-1}$ and $z = a_{n/2} + a_{n/2+1}x + \ldots + a_{n-1}x^{n/2-1}$, and returns two values: $f(x) = y + x^{n/2}z$, and x^n. Its time complexity is $\Theta(n)$.

5.39. Give a divide and conquer algorithm to solve the 1-dimensional closest pair problem: Given a set of n points on the x-axix, determine the two points that are closest to each other. Your algorithm should run in $O(n \log n)$ time.

The algorithm is shown below as Algorithm CLOSESTPAIRONED. It is assumed that n is a power of 2.

Algorithm 5.24 POLYNOMIAL
Input: An array $A[1..n]$ of n elements corresponding to $a_0, a_1, \ldots, a_{n-1}$ and x.
Output: $a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$

```
1. $p \leftarrow poly(A, 1, n, b)$
2. return $p$
```

Algorithm $poly(A, low, high, x)$

```
1. if $low = high$ then return $(A[low], x)$
2. else
3.     $mid \leftarrow \lfloor (low + high)/2 \rfloor$
4.     $(y, x_0) \leftarrow poly(A, low, mid, x)$
5.     $(z, x_1) \leftarrow poly(A, mid + 1, high, x)$
6.     $w \leftarrow y + x_1 z$
7.     return $(w, x_1^2)$
8. end if
```

Algorithm 5.25 CLOSESTPAIRONED
Input: A set S of n points on the x-axix.
Output: The minimum separation realized by two points in S.

```
1. $\delta \leftarrow cp(S)$
2. return $\delta$
```

Algorithm $cp(S)$

```
1. if $|S| = 2$ then return $|x_1 - x_2|$
2. Let $m$ be the median of $S$
3. Partition $S$ into $S_1$ of points $\leq m$ and $S_2$ of points $> m$
4. Recursively find the minimum separation $\delta_1$ in $S_1$ and the minimum
   separation $\delta_2$ in $S_2$
5. Let $x_1 = \max S_1$ and $x_2 = \min S_2$
6. Set $\delta_3 = x_2 - x_1$
7. Let $\delta = \min\{\delta_1, \delta_2, \delta_3\}$
8. return $\delta$
```

5.40. 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.

(a) Every point to the left of the vertical line L is compared with a fixed number of points to its right.

(b) Six points to the right of L are compared with every point to its left. See Fig. 5.11.

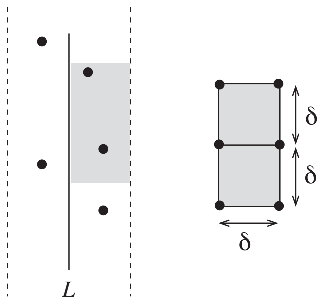

Fig. 5.11 Exrcise 5.40.

5.41. Rewrite the algorithm for the closest pair problem without making use of Algorithm MERGESORT. Instead, use a presorting step in which the input is sorted by y-coordinates at the start of the algorithm once and for all. The time complexity of your algorithm should be $\Theta(n \log n)$.

The algorithm is shown below as Algorithm CLOSESTPAIRS2.

5.42. Modify Algorithm DOMINANCE in Sec. 5.12 to solve the more general case, in which some points may lie on the same vertical line.

The solution to the case where some points may lie on the same vertical line is as follows. If in the current recursive call all points have the same x-coordinate, then none of them dominate the other, so set *count*(p) to zero for all of these points. This includes the case where there is only one point. The detailed algorithm is shown below as Algorithm DOMINANCE2.

5.43. Give a divide and conquer algorithm for the dominance problem in Sec. 5.12, which starts by dividing the input into four parts instead

Algorithm 5.26 CLOSESTPAIRS2
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. $\delta \leftarrow cp(1, n)$
4. return $\delta$
```

Algorithm $cp(low, high)$

```
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 $|Y|$      {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 $\delta$
```

of two (see Fig 5.12).

The algorithm is similar to Algorithm DOMINANCE discussed in Sec. 5.12, except for the following. All points in part D are dominated by points in part B. None of the points in part A are dominated by points in part C, and vice-versa. Some of the points in part A are dominated by points in part B. Some of the points in part D are dominated by points in parts A and C. Some of the points in part C are dominated by points in part B.

```
Algorithm 5.27 DOMINANCE2
Input: A set $S$ of $n$ points in the plane.
Output: The number of points in $S$ dominated by each point $p$.
  1. Sort the points in $S$ in nondecreasing order of their $x$-coordinates.
  2. $dom(S)$
  3. return $count(p)$ for all points $p \in S$

Algorithm $dom(S)$
  1. Let $L$ be the vertical line passing by the median point.
  2. Let $S_m$ be the points in $S$ that are lying on $L$, that is, having the
     same $x$-coordinate as $L$
  3. if $S_m = S$ then
  4.     for all points $p \in S_m$ $count(p) \leftarrow 0$
  5. else
  6.     Divide $S$ into two parts $S_l$ and $S_r$ such that the points
         in $S_l$ are on or to the left of $L$, and all points in $S_r$ are to the
         right of $L$.
  7.     $dom(S_l)$;    $dom(S_r)$
  8.     $d \leftarrow 0$
  9.     Sweep a horizontal line bottom-up starting from the point
         in $S$ with lowest $y$-coordinate:
 10.     for each point $p \in S$
 11.         if $p \in S_l$ then $d \leftarrow d + 1$
 12.         else $count(p) = count(p) + d$
 13.     end if
 14. end if
```

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

See Algorithm IDENTICALBT below.

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

See Algorithm HEIGHT below.

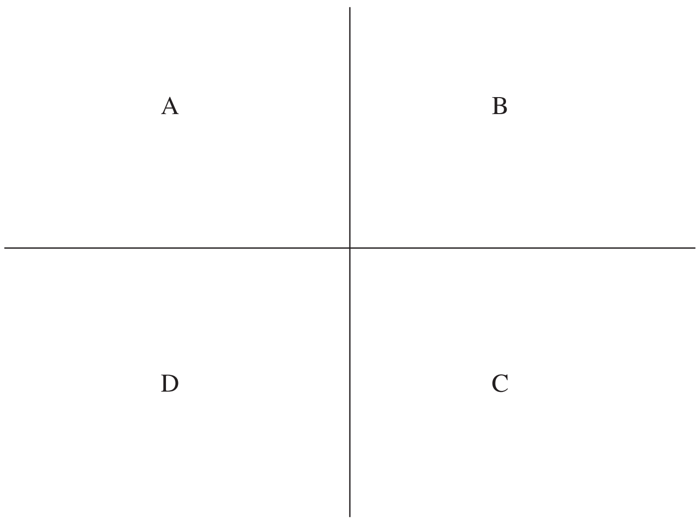

Fig. 5.12 Exercise 5.43.

Algorithm 5.28 IDENTICALBT
Input: Two binary trees: T_1 and T_2.

Output: **true** if the two trees are identical, and **false** otherwise

```
1. if $T_1$ is empty then if $T_2$ is empty
   then return true else return false
2. Let $T_3$ and $T_4$ be the left subtrees of $T_1$ and $T_2$.
3. Let $T_5$ and $T_6$ be the right subtrees of $T_1$ and $T_2$.
4. return IDENTICALBT($T_3, T_4$) and IDENTICALBT($T_5, T_6$)
```

Algorithm 5.29 HEIGHT
Input: A binary tree T.

Output: The height of T

```
1. if $T$ consists of one node then return 0 and exit
2. Let $T_1$ and $T_2$ be the left and right subtrees of $T$.
3. return $1 + \max\{$HEIGHT($T_1$), HEIGHT($T_2$)$\}$
```

6.10 Solutions

6.1. Prove Observation 6.1.

Let $A = a_1, a_2, \ldots, a_i$ and $B = b_1, b_2, \ldots, b_j$, and let $C = c_1, c_2, \ldots, c_k$ be a longest common subsequence of length l.
(a) It follows that $c_k = a_i = b_j$, for otherwise $a_i = b_j$ can be appended to C resulting in a common subsequence of length $> l$. Now, suppose there is an LCS of $a_1, a_2, \ldots, a_{i-1}$ and $b_1, b_2, \ldots, b_{j-1}$, say D, of length $> l - 1$. Then, by appending a_i to D, there would be an LCS of A and B of length $> l$, which is a contradiction.
(b) Suppose that $a_i \neq b_j$, and $L[i, j] > L[i, j-1]$. We show that $L[i, j] = L[i-1, j]$. Since $L[i, j] > L[i, j-1]$, it must be the case that $c_k = b_j$, which is not equal to a_i. But then, $L[i, j] = L[i-1, j]$.

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

The modified algorithm is shown as Algorithm LCS2 below. The algorithm returns the pair $(L[n, m], s)$, in which s is the desired subsequence.

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

Use two rows or two columns. Assume without loss of generality that $m \leq n$. Then, we may use two rows: the current row and the previous row. This is shown in Algorithm LCS3 below. Note that the work space needed is now $2 \times m = \Theta(m)$. The other case where $n \leq m$ is symmetrical.

6.4. Give a parenthesized expression for the optimal order of multiplying the five matrices in Example 6.4.

The parenthesized expression is $(M_1(M_2(M_3(M_4\ M_5))))$. For details of its derivation, see the solution to Exercise 6.7.

6.5. Consider applying Algorithm MATCHAIN on the following five matrices:

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

Algorithm 6.5 LCS2

Input: Two strings A and B of lengths n and m, over an alphabet Σ.

Output: The length of the longest common subsequence of A and B, and a subsequence of maximum length.

```
1.  for $i \leftarrow 0$ to $n$
2.      $L[i,0] \leftarrow 0$
3.  end for
4.  for $j \leftarrow 0$ to $m$
5.      $L[0,j] \leftarrow 0$
6.  end for
7.  $k \leftarrow 0$
8.  for $i \leftarrow 1$ to $n$
9.      for $j \leftarrow 1$ to $m$
10.         if $a_i = b_j$ then
11.             $L[i,j] \leftarrow L[i-1,j-1]+1$
12.             if $L[i,j] > k$ then
13.                 $k \leftarrow k+1$
14.                 $s[k] \leftarrow a_i$
15.             end if
16.         else $L[i,j] \leftarrow \max\{L[i,j-1], L[i-1,j]\}$
17.         end if
18.     end for
19. end for
20. return $(L[n,m], s)$
```

(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.

We apply the algorithm in the solution to Exercise 6.7.

(a) The array r is given by $r = 5, 10, 4, 6, 10, 2$. The cost matrix and the sequence matrix (δ matrix) are given by

$$\begin{pmatrix} 0 & 36 & 84 & 96 & 124 \\ 0 & 0 & 72 & 84 & 126 \\ 0 & 0 & 0 & 48 & 132 \\ 0 & 0 & 0 & 0 & 56 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} -1 & 1 & 2 & 1 & 4 \\ 0 & -1 & 2 & 2 & 4 \\ 0 & 0 & -1 & 3 & 4 \\ 0 & 0 & 0 & -1 & 4 \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

(b) The output of Procedure *matrixorder* is the parenthesized ex-

Algorithm 6.6 LCS3
Input: Two strings A and B of lengths n and m, over an alphabet Σ.

Output: The length of the longest common subsequence of A and B.

```
 1. $L[1,0] \leftarrow 0$;    $L[2,0] \leftarrow 0$
 2. for $j \leftarrow 0$ to $m$
 3.     $L[1,j] \leftarrow 0$
 4. end for
 5. for $i \leftarrow 1$ to $n$
 6.     for $j \leftarrow 1$ to $m$
 7.         if $a_i = b_j$ then $L[2,j] \leftarrow L[1,j-1]+1$
 8.         else $L[2,j] \leftarrow \max\{L[2,j-1], L[1,j]\}$
 9.         end if
10.     end for
11.     for $j \leftarrow 0$ to $m$
12.         $L[1,j] \leftarrow L[2,j]$
13.     end for
14. end for
15. return $L[2,m]$
```

pression ((1(2(3 4)))5), which corresponds to the multiplication $((M_1(M_2(M_3M_4)))M_5)$.

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

Let the three matrices be A, B and C with dimensions $a \times b, b \times c$ and $c \times d$. There are two possible multiplications: $A(BC)$, which costs $abd + bcd$, and $(AB)C$, which costs $abc + acd$. Set $abd + bcd \geq 100(abc + acd)$, and let $a = c = 1$. Then, $2bd \geq 100b + 100d$. Setting $b = d = 10^4$ yields $2 \times 10^8 = 2bd \geq 100b + 100d = 2 \times 10^6$. Hence, the desired dimensions are $a = c = 1$ and $b = d = 10^4$.

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

The modified algorithm is shown as Algorithm MATCHAIN2 below. In the modified algorithm, Step 18 has been added. k splits the chain of matrices into two sequences $M_1M_2 \ldots M_{k-1}$ and $M_kM_{k+1} \ldots M_n$ such that the optimal product consists of performing the two products $M' = M_1M_2 \ldots M_{k-1}$ and $M'' = M_kM_{k+1} \ldots M_n$, and then the multiplication $M'M''$. Procedure *matrixorder* in the algorithm outputs the parenthesized expression.

Algorithm 6.7 MATCHAIN2

Input: An array $r[1..n+1]$ of positive integers corresponding to the dimensions of a chain of n matrices, where $r[1..n]$ are the number of rows in the n matrices and $r[n+1]$ is the number of columns in M_n.

Output: The least number of scalar multiplications required to multiply the n matrices, and the order of their multiplication.

```
 1. for $i \leftarrow 1$ to $n$      {Fill in diagonal $d_0$}
 2.     $C[i,i] \leftarrow 0$
 3. end for
 4. for $i \leftarrow 1$ to $n$
 5.     for $j \leftarrow 1$ to $n$
 6.         if $i = j$ then $\delta[i,j] \leftarrow -1$ else $\delta[i,j] \leftarrow 0$
 7.     end for
 8. end for
 9. for $d \leftarrow 1$ to $n-1$      {Fill in diagonals $d_1$ to $d_{n-1}$}
10.     for $i \leftarrow 1$ to $n-d$      {Fill in entries in diagonal $d_i$}
11.         $j \leftarrow i+d$
12.         comment: The next three lines compute $C[i,j]$
13.         $C[i,j] \leftarrow \infty$
14.         for $k \leftarrow i+1$ to $j$
15.             $temp \leftarrow C[i,k-1] + C[k,j] + r[i]r[k]r[j+1]$
16.             if $temp < C[i,j]$ then
17.                 $C[i,j] \leftarrow temp$
18.                 $\delta[i,j] = k-1$
19.             end if
20.         end for
21.     end for
22. end for
23. output $C[1,n]$
24. $order \leftarrow \{\}$
25. output $matrixorder(1,n)$
```

Algorithm *matrixorder*(*low*, *high*)

```
1. if $low = high$ then
2.     Append $low$ to $order$ and return $order$
3. Append "(" to $order$
4. $s \leftarrow \delta[low, high]$
5. $matrixorder(low, s)$
6. $matrixorder(s+1, high)$
7. Append ")" to $order$
8. return $order$
9. end if
```

Now, we apply the above algorithm on the instance in Example 6.4. In this example, we have

$M_1 : 5\times 10,\quad M_2 : 10\times 4,\quad M_3 : 4\times 6,\quad M_4 : 6\times 10,\quad M_5 : 10\times 2.$

Thus, the array r is given by $r = 5, 10, 4, 6, 10, 2$. The cost matrix and the sequence matrix (δ matrix) are given by

$$\begin{pmatrix} 0 & 200 & 320 & 620 & 348 \\ 0 & 0 & 240 & 640 & 248 \\ 0 & 0 & 0 & 240 & 168 \\ 0 & 0 & 0 & 0 & 120 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} -1 & 1 & 2 & 3 & 1 \\ 0 & -1 & 2 & 2 & 2 \\ 0 & 0 & -1 & 3 & 3 \\ 0 & 0 & 0 & -1 & 4 \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

The output of Procedure *matrixorder* is the parenthesized expression (1(2(3(4 5)))), which corresponds to the multiplication $(M_1(M_2(M_3(M_4\ M_5))))$.
As another example, consider the chain of matrices:

$M_1 : 10\times 15, M_2 : 15\times 5, M_3 : 5\times 15, M_4 : 15\times 10, M_5 : 10\times 20, M_6 : 20\times 10.$

Thus, the array r is given by $r = 10, 15, 5, 15, 10, 20, 10$. The cost matrix and the sequence matrix (δ matrix) are given by

$$\begin{pmatrix} 0 & 15750 & 7875 & 9375 & 11875 & 15125 \\ 0 & 0 & 2625 & 4375 & 7125 & 10500 \\ 0 & 0 & 0 & 750 & 2500 & 5375 \\ 0 & 0 & 0 & 0 & 1000 & 3500 \\ 0 & 0 & 0 & 0 & 0 & 5000 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} -1 & 1 & 2 & 2 & 2 & 2 \\ 0 & -1 & 2 & 2 & 2 & 2 \\ 0 & 0 & -1 & 3 & 4 & 5 \\ 0 & 0 & 0 & -1 & 4 & 4 \\ 0 & 0 & 0 & 0 & -1 & 5 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

The output of Procedure *matrixorder* is the parenthesized expression ((1 2)(((3 4)5)6)) which corresponds to the multiplication $((M_1M_2)(((M_3M_4)M_5)M_6))$.

6.8. 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.

Let G be any directed graph with no negative cycles.

6.9. 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.

Let G be any directed graph with at least one negative cycle.

6.10. 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).

After running Algorithm FLOYD, check whether there is a negative number in the diagonal of the output matrix. There is a negative cycle if and only if there is such a number.

6.11. Prove Observation 6.2.

In an optimal packing of $u_1, u_2, \ldots, u_i$, either u_i is in the optimal packing or not. If it is, then the rest of the knapsack of size $j - s_i$ must constitute an optimal packing of (some of) the items in $u_1, u_2, \ldots, u_{i-1}$. If it is not, then the knapsack of size j must constitute an optimal packing of the items in $u_1, u_2, \ldots, u_{i-1}$.

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

The values matrix is given by

$$\begin{pmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&4&4&4&4&4&4&4&4&4&4&4&4&4&4&4&4&4&4&4&4\\
0&0&4&4&6&6&6&10&10&10&10&10&10&10&10&10&10&10&10&10&10&10\\
0&0&4&4&6&6&7&10&10&11&11&13&13&13&17&17&17&17&17&17&17&17\\
0&0&4&4&6&6&7&10&10&11&13&13&15&15&17&19&19&20&20&22&22&22\\
0&0&4&4&6&6&7&10&10&11&13&14&15&16&17&19&20&20&21&23&23&25
\end{pmatrix}.$$

The packed items are $2, 4$ and 5 of sizes $5, 8$ and 9. The total value is 25.

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

In this case, the statement:
if $s_i \leq j$ **then** $V[i,j] \leftarrow \max\{V[i,j], V[i-1, j-s_i] + v_i\}$

will fail to execute, as there will be a reference outside the rectangular array V.

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

Use two rows: the current row and the previous row. This is shown in Algorithm KNAPSACK2 below. Note that the work space needed is now $2 \times C = \Theta(C)$.

Algorithm 6.8 KNAPSACK2
Input: A set of items $U = \{u_1, u_2, \dots, u_n\}$ with sizes $s_1, s_2, \dots, s_n$ and values $v_1, v_2, \dots, v_n$ and a knapsack capacity C.
Output: The maximum value of the function $\sum_{u_i \in S} v_i$ subject to $\sum_{u_i \in S} s_i \leq C$ for some subset of items $S \subseteq U$.

```
 1. $V[1,0] \leftarrow 0$;   $V[2,0] \leftarrow 0$
 2. for $j \leftarrow 0$ to $C$
 3.     $V[1,j] \leftarrow 0$
 4. end for
 5. for $i \leftarrow 1$ to $n$
 6.     for $j \leftarrow 1$ to $C$
 7.         $V[2,j] \leftarrow V[1,j]$
 8.         if $s_i \leq j$ then $V[2,j] \leftarrow \max\{V[2,j], V[1,j-s_i]+v_i\}$
 9.     end for
10.     for $j \leftarrow 0$ to $m$
11.         $V[1,j] \leftarrow V[2,j]$
12.     end for
13. end for
14. return $V[2,C]$
```

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

The modified algorithm is shown below as Algorithm KNAPSACK3. It makes use of a rectangular array K to store triplets (w, x, y), where w is an item used in the packing for value $V[i, j]$, and x and y point to the last items in the packing for value $V[i-1, j-s_i]$. The **while** loop at the end of the algorithm is used to retrieve the items in an optimal packing.

Algorithm 6.9 KNAPSACK3

Input: A set of items $U = \{u_1, u_2, \ldots, u_n\}$ with sizes $s_1, s_2, \ldots, s_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack capacity C.

Output: The maximum value of the function $\sum_{u_i \in S} v_i$ subject to $\sum_{u_i \in S} s_i \leq C$ for some subset of items $S \subseteq U$, and a subset of items of maximum value.

```
1.  for $i \leftarrow 0$ to $n$
2.      $V[i,0] \leftarrow 0$
3.      $K[i,0] \leftarrow \{0,0,0\}$
4.  end for
5.  for $j \leftarrow 0$ to $C$
6.      $V[0,j] \leftarrow 0$
7.      $K[0,j] \leftarrow \{0,0,0\}$
8.  end for
9.  for $i \leftarrow 1$ to $n$
10.     for $j \leftarrow 1$ to $C$
11.         $V[i,j] \leftarrow V[i-1,j]$
12.         $K[i,j] \leftarrow \{0, i-1, j\}$
13.         if $s_i \leq j$ then
14.             $temp \leftarrow V[i-1, j-s_i] + v_i$
15.             if $temp \geq V[i,j]$ then
16.                 $V[i,j] \leftarrow temp$
17.                 $K[i,j] \leftarrow \{i, i-1, j-s_i\}$
18.             end if
19.         end if
20.     end for
21. end for
22. output $V[n,C]$
23. $i \leftarrow n;\ j \leftarrow C$
24. while $i > 0$ and $j > 0$
25.     $(w,i,j) \leftarrow K[i,j]$
26.     if $w > 0$ then output $w$
27. end while
```

6.16. 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. 6.6. This technique is called *scaling and rounding* (see Sec. 14.6). 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.

The running time will be $\Theta(nC/K)$. Consider the instance: $n = 2, s_1 = v_1 = 14, s_2 = v_2 = 15, C = 20$. The optimal solution is to pack one item of size s_2. However, using $K = 10$, the new instance is $n = 2, s'_1 = 1, s'_2 = 1, C = 2$, with optimal solution of packing both items. This instance provides a counterexample.

6.17. 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 $1 \le j \le n$, 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 $\sum_{i=1}^{n} x_i s_i \le C$. $x_1, x_2, \ldots, x_n$ are nonnegative integers. Note that $x_j = 0$ means that no item of the jth type is packed in the knapsack. Rewrite the dynamic programming algorithm for this version of the knapsack problem.

The modified algorithm is shown as Algorithm KNAPSACK4 below.

6.18. Solve the following instance of the version of the knapsack problem described in Exercise 6.17. 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.

The values matrix is given below. Thus, the optimal total value is 18. This is achieved by packing one item of the first type and two items of the second type.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 8 & 8 & 12 & 12 & 16 \\ 0 & 0 & 4 & 7 & 8 & 11 & 14 & 15 & 18 \\ 0 & 0 & 4 & 7 & 8 & 11 & 14 & 15 & 18 \\ 0 & 0 & 4 & 7 & 8 & 11 & 14 & 15 & 18 \end{pmatrix}.$$

6.19. Show how to modify the knapsack algorithm discussed in Exer-

Algorithm 6.10 KNAPSACK4
Input: A set of types of items items $U = \{u_1, u_2, \ldots, u_n\}$ with sizes $s_1, s_2, \ldots, s_n$ and
values $v_1, v_2, \ldots, v_n$ and a knapsack capacity C.
Output: The maximum value of the function $\sum_{u_i \in S} x_i v_i$ subject to
$\sum_{u_i \in S} x_i s_i \leq C$ for some subset of items $S \subseteq U$.

```
 1. for $i \leftarrow 0$ to $n$
 2.     $V[i, 0] \leftarrow 0$
 3. end for
 4. for $j \leftarrow 0$ to $C$
 5.     $V[0, j] \leftarrow 0$
 6. end for
 7. for $i \leftarrow 1$ to $n$
 8.     for $j \leftarrow 1$ to $C$
 9.         $V[i, j] \leftarrow V[i-1, j]$
10.         if $s_i \leq j$ then
11.             for $k \leftarrow 1$ to $\lfloor j/s_i \rfloor$
12.                 $V[i, j] \leftarrow \max\{V[i, j], V[i-1, j-ks_i] + kv_i\}$
13.             end for
14.         end if
15.     end for
16. end for
17. return $V[n, C]$
```

cise 6.17 so that it computes the number of items packed from each type.

The modified algorithm is shown below as Algorithm KNAPSACK5. It makes use of a rectangular array K to store the 4-tuples (w, k, x, y), where w is an item used in the packing for value $V[i, j]$, k is the number of items of w packed, and x and y point to the last items in the packing for value $V[i-1, j-ks_i]$. The **while** loop at the end of the algorithm is used to retrieve the items and their numbers in an optimal packing.

6.20. 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).

Algorithm 6.11 KNAPSACK5

Input: A set of types of items items $U = \{u_1, u_2, \ldots, u_n\}$ with sizes $s_1, s_2, \ldots, s_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack capacity C.

Output: The maximum value of the function $\sum_{u_i \in S} x_i v_i$ subject to $\sum_{u_i \in S} x_i s_i \leq C$ for some subset of items $S \subseteq U$, and a subset of the items.

```
 1. for $i \leftarrow 0$ to $n$
 2.     $V[i, 0] \leftarrow 0$
 3.     $K[i, 0] \leftarrow \{0, 0, 0, 0\}$
 4. end for
 5. for $j \leftarrow 0$ to $C$
 6.     $V[0, j] \leftarrow 0$
 7.     $K[0, j] \leftarrow \{0, 0, 0, 0\}$
 8. end for
 9. for $i \leftarrow 1$ to $n$
10.     for $j \leftarrow 1$ to $C$
11.         $V[i, j] \leftarrow V[i-1, j]$
12.         $K[i, j] \leftarrow \{0, 0, i-1, j\}$
13.         if $s_i \leq j$ then
14.             for $k \leftarrow 1$ to $\lfloor j/s_i \rfloor$
15.                 $temp \leftarrow V[i-1, j-ks_i] + kv_i$
16.                 if $temp \geq V[i, j]$ then
17.                     $V[i, j] \leftarrow temp$
18.                     $K[i, j] \leftarrow \{i, k, i-1, j-ks_i\}$
19.                 end if
20.             end for
21.         end if
22.     end for
23. end for
24. output $V[n, C]$
25. $i \leftarrow n$; $j \leftarrow C$
26. while $i > 0$ and $j > 0$
27.     $(w, k, i, j) \leftarrow K[i, j]$
28.     if $k > 0$ then output $(w, k)$
29. end while
```

(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 6.17? Explain.

(a) The dynamic programming algorithm is given below as Algorithm MONEYCHANGEDP. It makes use of a rectangular array K to store the 4-tuples (w, k, x, y), where w is a coin used in the change for value $V[i, j]$, k is the number of coins w given in the change, and x and y point to the last solution for the change of value $V[i-1, j-kv_i]$. The **while** loop at the end of the algorithm is used to retrieve the coins and their numbers in an optimal change.

Algorithm 6.12 MONEYCHANGEDP
Input: A set of n coins with values $v_1, v_2, \ldots, v_n$, where $v_1 = 1$ and a value y.
Output: Pay value y using minimum number of coins. Output set of coins.

```
 1. for $j \leftarrow 1$ to $C$
 2.     $V[1, j] \leftarrow \lfloor j/v_1 \rfloor$
 3.     $K[1, j] \leftarrow \{1, \lfloor j/v_1 \rfloor, 0, 0\}$
 4. end for
 5. for $i \leftarrow 2$ to $n$
 6.     for $j \leftarrow 1$ to $C$
 7.         $V[i, j] \leftarrow V[i-1, j]$
 8.         $K[i, j] \leftarrow \{0, 0, i-1, j\}$
 9.         if $v_i \leq j$ then
10.             for $k \leftarrow 1$ to $\lfloor j/v_i \rfloor$
11.                 $j_1 \leftarrow j - kv_i$
12.                 $temp \leftarrow V[i-1, j_1] + \mathrm{k}$
13.                 if $temp \leq V[i, j]$ then
14.                     $V[i, j] \leftarrow temp$
15.                     $K[i, j] \leftarrow \{i, k, i-1, j_1\}$
16.                 end if
17.             end for
18.         end if
19.     end for
20. end for
21. output $V[n, C]$
22. $i \leftarrow n;\ j \leftarrow C$
23. while $i > 0$ and $j > 0$
24.     $(w, k, i, j) \leftarrow K[i, j]$
25.     if $k > 0$ then output $(w, k)$
26. end while
```

(b) The time complexity of the algorithm is $\Theta(n, y)$, which is also the space complexity.

(c) The algorithm is very similar to Algorithm KNAPSACK5.

6.21. Apply the algorithm in Exercise 6.20 to the instance $v_1 = 1, v_2 = 5, v_3 = 7, v_4 = 11$ and $y = 20$.

The values matrix is given by

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 5 & 2 & 3 & 4 & 5 & 6 & 3 & 4 & 5 & 6 & 7 & 4 \\ 1 & 2 & 3 & 4 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 & 4 & 3 & 4 & 3 & 4 \\ 1 & 2 & 3 & 4 & 1 & 2 & 1 & 2 & 3 & 2 & 1 & 2 & 3 & 2 & 3 & 2 & 3 & 2 & 3 & 4 \end{pmatrix}.$$

The change is $\{(1,2),(3,1),(4,1)\}$. That is, two coins of value $v_1 = 1$, one coin of value $v_3 = 7$ and one coin of value $v_4 = 11$.

6.22. 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).

The algorithm, shown below as Algorithm TRANSCLOSURE, is a simple modification of Algorithm FLOYD.

Algorithm 6.13 TRANSCLOSURE
Input: An $n \times n$ matrix $A[1..n, 1..n]$ such that $A[i, j] = 1$ if there is an edge (i, j) in a directed graph $G = (\{1, 2, \ldots, n\}, E)$, and 0 otherwise.
Output: A^*, the transitive closure of A.

```
1. $A^* \leftarrow A$      {copy the input matrix $A$ into $A^*$}
2. for $k \leftarrow 1$ to $n$
3.     for $i \leftarrow 1$ to $n$
4.         for $j \leftarrow 1$ to $n$
5.             $A^*[i, j] = A^*[i, j] \vee (A^*[i, k] \wedge A^*[k, j])$
6.         end for
7.     end for
8. end for
```

6.23. 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).

The algorithm, shown below as Algorithm SHORTESTDIST2, is a simple modification of Algorithm FLOYD.

Algorithm 6.14 SHORTESTDIST2

Input: An $n \times n$ matrix $A[1..n, 1..n]$ such that $A[i, j]$ is 1 if there is an edge (i, j) in a directed graph $G = (\{1, 2, \ldots, n\}, E)$, and 0 otherwise.

Output: A matrix D with $D[i, j]$ = the distance from i to j measured in number of edges.

```
1. $D \leftarrow A$      {copy the input matrix $A$ into $D$}
2. for $k \leftarrow 1$ to $n$
3.     for $i \leftarrow 1$ to $n$
4.         for $j \leftarrow 1$ to $n$
5.             $D[i, j] = \min\{D[i, j], D[i, k] + D[k, j]\}$
6.         end for
7.     end for
8. end for
```

6.24. 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?

A longest path between s and t is the same as a shortest path in the graph G' derived from G by changing every weight to its negation. Therefore, apply any algorithm for the shortest path problem on G' to find the longest path from s to all other vertices. The time complexity is the same as that for the shortest path algorithm. If we use Algorithm FLOYD, then the time complexity is $\Theta(n^3)$.

7.9 Solutions

7.1. Suppose in the money change problem of Example 7.1, 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 $y < 2^{k+1}$.

Let the binary representation of y be $b_0, b_1, \ldots, b_{k-1}$. For $0 \leq i \leq k-1$, include value 2^i if and only if $b_i = 1$.

7.2. 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.

Consider the graph shown in Fig. 7.10. Using this algorithm, vertex a will be added to the cover first. The resulting vertex cover is $\{a, b, c, d\}$, while the minimum vertex cover is $\{b, c, d\}$.

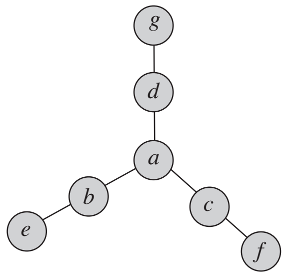

Fig. 7.10 Undirected graph for Exercise 7.2.

7.3. 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.

Consider the graph shown in Fig. 7.11. Deleting vertex a will not result in a maximum clique.

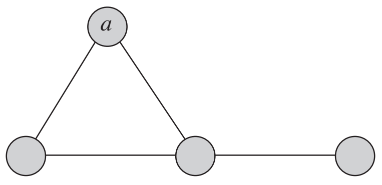

Fig. 7.11 Undirected graph for Exercise 7.3.

7.4. 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.

Consider the graph shown in Fig. 7.12, and the coloring shown. The number of colors used by the algorithm is 4, while the minimum number of colors is 3.

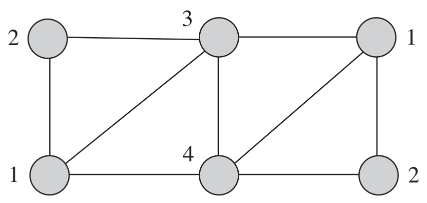

Fig. 7.12 Undirected graph for Exercise 7.4.

7.5. 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).

Use an algorithm similar to Algorithm HUFFMAN. Merge the two smallest arrays, then the next two smallest arrays, etc. See Exercise 7.6.

7.6. Analyze the time complexity of the algorithm in Exercise 7.5. Hint: If we merge the two smallest arrays, then the next two smallest arrays, etc., then each time we merge, the resulting array size is at least double the smaller array size.

If we merge the two smallest arrays, then the next two smallest arrays, etc., then each time we merge, the resulting array size is at least double the smaller array size. Thus, the number of times an item participates in a comparison (i.e. is compared) is at most $O(\log n)$. This means the total number of comparisons is $O(n \log n)$.

7.7. 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).

Consider the directed graph shown in Fig. 7.13. Starting from vertex s, the algorithm will first visit x, then from x, it will visit t.

The length of the path found is 4, while the shortest path from s to t is of length 2.

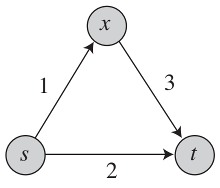

Fig. 7.13 Directed graph for Exercise 7.7.

7.8. Modify Algorithm DIJKSTRA so that it finds the shortest paths in addition to their lengths. Hint: Use an array P such that $P[v]$ stores the parent of the new vertex v.

In Algorithm DIJKSTRA, add the statement $P[w] \leftarrow y$ after the statement $\lambda[w] \leftarrow \lambda[y] + \mathit{length}[y, w]$.

7.9. Prove that the subgraph defined by the paths obtained from the modified shortest path algorithm as described in Exercise 7.8 is a tree. This tree is called the *shortest path tree.*

$P[w]$ as defined in Exercise 7.8 is a single value, and since all vertices except the start vertex are assigned a parent, the number of edges in the subgraph is $n - 1$. Obviously, the subgraph is connected. Since it is connected and consists of $n - 1$ edges, it follows that it is a tree.

7.10. Can a directed graph have two distinct shortest path trees (see Exercise 7.9)? Prove your answer.

Consider the directed graph G shown in Fig. 7.14. Starting from vertex s, there are two shortest path trees shown in the figure: T_1 and T_2.

7.11. Show that the proof of correctness of Algorithm DIJKSTRA (Lemma 7.1) does not work if some of the edges in the input graph have negative weights.

If there are negative weights, the statement $\lambda[y] \leq \lambda[x] +$

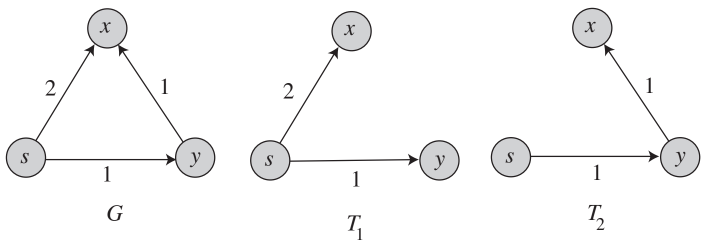

Fig. 7.14 Directed graph and trees for Exercise 7.10.

length(x, w) may imply that $\lambda[y] < \lambda[x]$. This contradicts the assumption that x left Y before y.

7.12. 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.

Since removing directions results in a planar graph, $m = O(n)$. Hence, the running time is $O(n \log n)$. This is to be contrasted to the performance of Algorithm DIJKSTRA, which is $O(n^2)$.

7.13. 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 $O(m)$?

Use d-heaps, where $d = \lceil 2 + m/n \rceil = \lceil 2 + n^{0.2} \rceil$.

7.14. 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.

Algorithm KRUSKAL costs $O(m \log m) = O(n^{1.99} \log n)$. Algorithm PRIM costs $O(n^2)$. Since $n^{1.99} \log n = o(n^2)$, Kruscal's algorithm is more efficient.

7.15. Refer to Fig. 7.15.

(a) Run Prim's algorithm starting from vertex 3 in order to find a minimum spanning tree of the graph. Make sure you indicate the order in which each edge is added to the minimum spanning tree.

(b) Can this graph have more than one minimum spanning trees? Justify your answer.

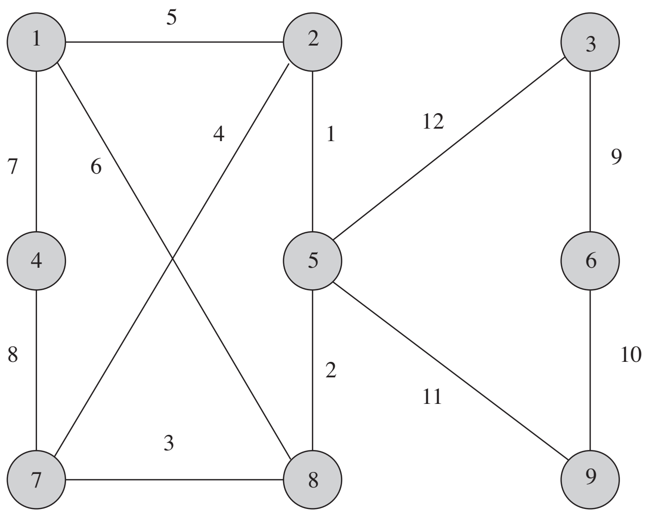

Fig. 7.15 Undirected graph.

(a) Similar to the example shown in Fig. 7.5.
(b) There is only one minimum spanning tree since the costs are distinct. See Exercise 7.18.

7.16. 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.

Let T be a minimum spanning tree. If e is not in T, add e to T. This will create a cycle C, which contains e, and at least one more edge e'. Now, the weight of e is $\leq$ the weight of e', since e is of minimum weight. If we delete e' from $T \cup \{e\}$, the resulting graph is still connected and is a tree. Also, the total weight of this new tree, which contains e, is less than or equal to that of T, and thus it is of minimum total weight.

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

It works correctly with negative weights. To see why, add a large positive number to all weights. Then, the resulting minimum spanning tree is the same.

7.18. 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.

Refer to Fig. 7.16 for an illustration. Let G be a connected graph with two minimum spanning trees T and T'. We will show that G contains two edges that have the same weight. Each of the two spanning trees must contain an edge that the other tree does not include. Let e be a minimum-weight edge in $T - T'$, and let e' be a minimum-weight edge in $T' - T$. Without loss of generality, suppose $w(e) \leq w(e')$, where w is the weight function. The subgraph $T' \cup \{e\}$ contains a unique cycle C, which passes through the edge e. Let e'' be any edge of this cycle that is not in T (e'' may be equal to e'). Since $e \in T$, we must have $e'' \neq e$, and therefore $e'' \in T' - T$. It follows that $w(e'') \geq w(e') \geq w(e)$. Now, consider the spanning tree $T'' = T' + e - e''$ (T'' may be equal to T). We immediately have $w(T'') = w(T') + w(e) - w(e'') \leq w(T')$. But T' is a minimum spanning tree, so we must have $w(T'') = w(T')$; in other words, T'' is also a minimum spanning tree. It follows that $w(e) = w(e'')$.

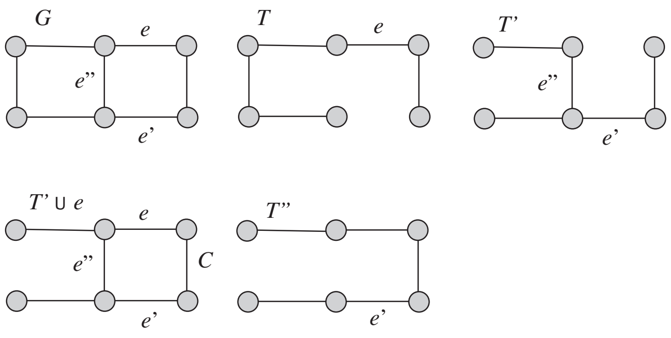

Fig. 7.16 Illustration of the proof for Exercise 7.18.

7.19. 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 on three vertices, is 3.

The number of spanning trees for n distinct vertices is n^{n-2}.

7.20. 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 7.9). 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 $T = T'$.

T (after removing directions) and T' may be different. Consider the directed graph G shown in Fig. 7.17. Starting from vertex s, T_1 is a shortest path tree. If we remove the directions in G, and find the minimum spanning tree, the resulting tree T_2 is different from T_1 after removing directions.

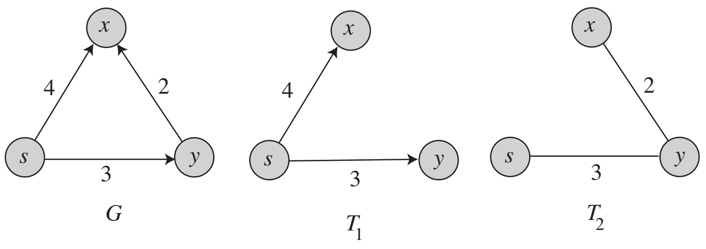

Fig. 7.17 Directed graph and trees for Exercise 7.20.

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

Number of added edges = total number of edges = $2(n-1) = 2n-2$. Number of added vertices = $n - 1$, and hence total number of vertices = $n + (n - 1) = 2n - 1$. Since the total number of edges is equal to the total number of vertices minus one, and the graph is obviously connected, it follows that it is a tree.

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

The principle of optimality holds, and hence it is a dynamic programming algorithm.

7.23. 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 b_i dominates w_j. A matching $M = \{(b_{i_1}, w_{j_1}), (b_{i_2}, w_{j_2}), \ldots, (b_{i_k}, w_{j_k})\}$ between the black and white points is maximum if k, the number of matched pairs in M, is maximum. Design a greedy algorithm to find a maximum matching in $O(n \log n)$ time. (Hint: Sort the black points in increasing x-coordinates and use a heap for the white points).

Consider the following greedy algorithm that finds a maximum matching. Sort the black points in increasing x-coordinates and, let this sorted list be B_s. Scan the points in B_s in increasing order of their x-coordinates, each time matching the current black point b with the unmatched white point of maximum y-coordinate that is dominated by b. Use a heap to store the white points. Clearly, the running time is $O(n \log n)$.

8.9 Solutions

8.1. Give an iterative version of Algorithm DFS that uses a stack to store unvisited vertices.

An iterative version of depth-first search is given below as Algorithm DFSITERATIVE. In the algorithm, *St* is a stack used to store unvisited vertices.

Algorithm 8.5 DFSITERATIVE
Input: A directed or undirected graph $G = (V, E)$.

Output: Numbering of the vertices in depth-first search order.

```
 1. $predfn \leftarrow 0$;    $postdfn \leftarrow 0$
 2. for each vertex $v \in V$
 3.     mark $v$ unvisited
 4. end for
 5. for each vertex $v \in V$
 6.     if $v$ is marked unvisited then $dfs(v)$
 7. end for
```

Algorithm *dfs*(v)

```
 1. $St \leftarrow \{v\}$
 2. mark $v$ visited
 3. while $St \neq \{\}$
 4.     $v \leftarrow Pop(St)$
 5.     $predfn \leftarrow predfn + 1$
 6.     for each edge $(v, w) \in E$
 7.         if $w$ is marked unvisited then
 8.             Push$(w, St)$
 9.             mark $w$ visited
10.         end if
11.     end for
12.     $postdfn \leftarrow postdfn + 1$
13. end while
```

8.2. What will be the time complexity of the depth-first search algorithm if the input graph is represented by an adjacency matrix (see Sec. 2.3.1 for graph representation).

The running time will be $\Theta(n)$ for each vertex, as the algorithm needs to search for the next vertex to be visited in the adjacency matrix. Thus, the time complexity is $\Theta(n^2)$.

8.3. Show that when depth-first search is applied to an undirected graph G, the edges of G will be classified as either tree edges or back edges. That is, there are no forward edges or cross edges.

An edge (v, w) in an undirected graph is a tree edge if w was first visited when exploring the edge (v, w), otherwise it is a back edge.

8.4. Suppose that Algorithm DFS is applied to an undirected graph G. Give an algorithm that classifies the edges of G as either tree edges or back edges.

Let $f(v)$ be *predfn* when v was visited in the call *dfs*(v) (see Algorithm DFS2). Then, (v, w) is a tree edge if $f(v) < f(w)$, otherwise it is a back edge.

Algorithm 8.6 DFS2
Input: A (directed or undirected) graph $G = (V, E)$.
Output: Preordering and postordering of the vertices in the corresponding depth-first search tree.

```
1. predfn ← 0;    postdfn ← 0
2. for each vertex $v \in V$
3.     mark $v$ unvisited
4.     $f(v) \leftarrow 0$
5.     $g(v) \leftarrow 0$
6. end for
7. for each vertex $v \in V$
8.     if $v$ is marked unvisited then dfs($v$)
9. end for
```

Algorithm *dfs*(v)

```
1. mark $v$ visited
2. predfn ← predfn + 1
3. $f(v) \leftarrow$ predfn
4. for each edge $(v, w) \in E$
5.     if $w$ is marked unvisited then dfs($w$)
6. end for
7. postdfn ← postdfn + 1
8. $g(v) \leftarrow$ postdfn
```

8.5. Suppose that Algorithm DFS is applied to a directed graph G. Give an algorithm that classifies the edges of G as either tree edges, back edges, forward edges or cross edges.

Let $f(v)$ and $g(v)$ be *predfn* and *postdfn*, respectively, when v was visited in the call *dfs*(v) (see Algorithm DFS2). Then,
(1) (v, w) is a tree edge if $f(v) < f(w), g(w) < g(v)$ and w was first visited when exploring the edge (v, w).
(2) (v, w) is a forward edge if $f(v) < f(w), g(w) < g(v)$ and w was marked visited when exploring the edge (v, w).
(3) (v, w) is a back edge if $f(w) < f(v)$ and $g(v) < g(w)$.
(4) (v, w) is a cross edge, otherwise.
See Fig. 8.2 for an example.

8.6. Give an algorithm that counts the number of connected components in an undirected graph using depth-first search or breadth-first search.

See Algorithm CONNECTEDCOMP1 below.

Algorithm 8.7 CONNECTEDCOMP1
Input: An undirected graph $G = (V, E)$.
Output: Number of connected components in G.

```
 1. $cc \leftarrow 0$
 2. for each vertex $v \in V$
 3.     mark $v$ unvisited
 4. end for
 5. for each vertex $v \in V$
 6.     if $v$ is marked unvisited then
 7.         $cc \leftarrow cc + 1$
 8.         dfs($v$)
 9.     end if
10. end for
11. return $cc$
```

Algorithm *dfs*(v)

```
 1. mark $v$ visited
 2. for each edge $(v, w) \in E$
 3.     if $w$ is marked unvisited then dfs($w$)
 4. end for
```

8.7. Given an undirected graph G, design an algorithm to list the vertices in each connected component of G separately.

See Algorithm CONNECTEDCOMP2 below. The algorithm lists the pairs (cc, v), where cc is a connected component number, and v is a vertex in connected component cc.

Algorithm 8.8 CONNECTEDCOMP2
Input: An undirected graph $G = (V, E)$.

Output: List of vertices and their connected components in G.

```
 1. $cc \leftarrow 0$
 2. for each vertex $v \in V$
 3.     mark $v$ unvisited
 4. end for
 5. for each vertex $v \in V$
 6.     if $v$ is marked unvisited then
 7.         $cc \leftarrow cc + 1$
 8.         dfs($v$)
 9.     end if
10. end for
```

Algorithm *dfs*(v)

```
1. mark $v$ visited
2. output $(cc, v)$
3. for each edge $(v, w) \in E$
4.     if $w$ is marked unvisited then dfs($w$)
5. end for
```

8.8. Give an $O(n)$ time algorithm to determine whether a connected undirected graph with n vertices contains a cycle.

Do at most $n + 1$ iterations of Algorithm DFS. If there is a cycle, then a vertex will be visited twice. Stop the search after $n + 1$ iterations or as soon as a vertex is visited for the second time. This is indicated by exploring a back edge. In this case, only $O(n)$ edges are explored, and hence the running time is $O(n)$.

8.9. Let T be the depth-first search tree resulting from a depth-first search traversal on a connected undirected graph. Show that the root of T is an articulation point if and only if it has two or more children. (See Sec. 8.3.3)

Let r be the root of the depth-first search tree T. First, suppose r is an articulation point. Then, the removal of r from G would cause the graph to be disconnected, so r has at least two children in T.

Now suppose r has at least two children u and v in T. Then, there is no path from u to v in G which doesn't go through r, since otherwise u would be an ancestor of v or v would be an ancestor of u. Thus, removing r disconnects the component containing u and the component containing v, so r is an articulation point.

8.10. Let T be the depth-first search tree resulting from a depth-first search traversal on a connected undirected graph. Show that a vertex v other than the root is an articulation point if and only if v has a child w with $\beta[w] \geq \alpha[v]$. (See Sec. 8.3.3 for definition of $\alpha[v]$ and $\beta[w]$).

Suppose that v is a vertex of the depth-first search tree different from the root, and that v is an articulation point. Then, v has a child w such that neither w nor any of w's descendants have back edges to a proper ancestor of v. In other words, for some child w of v we have $\beta[w] \geq \alpha[v]$.
Now suppose that v is not an articulation point, so for every child w of v, there exists a descendant of that child which has a back edge to a proper ancestor of v. In other words, for every child w of v we have $\beta[w] < \alpha[v]$.

8.11. An edge of a connected undirected graph G is called a *bridge* if its deletion disconnects G. Modify the algorithm for finding articulation points so that it detects bridges instead of articulation points.

An edge is a bridge if and only if both of its endpoints are articulation points or one endpoint is an articulation point and the other is a vertex of degree 1. So, run Algorithm ARTICPOINTS to find the articulation points, and then decide which edges are bridges.

8.12. Show the result of running breadth-first search on the undirected graph of Fig. 8.9 starting at vertex f.

Similar to the example in Fig. 8.7.

8.13. Show the result of running breadth-first search on the directed graph of Fig. 8.10 starting at vertex e.

Similar to the example in Fig. 8.7.

8.14. Show that when breadth-first search is applied to an undirected graph G, the edges of G will be classified as either tree edges or cross edges. That is, there are no back edges or forward edges.

Suppose there is a path from u to w, and there is a path from w to v. Suppose also that there is an edge (u, v). Moreover, let u be closer to the root than v. Then, when processing vertex u by breadth-first search, vertex v will be pushed, that is, the edge (u, v) will be considered immediately, not later as a forward or back edge. So, there are no forward or back edges since breadth-first search considers shortest paths first.

8.15. Show that when breadth-first search is applied to a directed graph G, the edges of G will be classified as tree edges, back edges or cross edges. That is, unlike the case of depth-first search, the search does not result in forward edges.

Suppose there is a path from u to w, and there is a path from w to v. Suppose also that there is an edge (u, v). Then, when processing vertex u by breadth-first search, vertex v will be pushed, that is, the edge (u, v) will be considered immediately, not later as a forward edge. So, there are no forward edges since breadth-first search considers shortest paths first.

8.16. Let G be a graph (directed or undirected), and let s be a vertex in G. Modify Algorithm BFS so that it outputs the shortest path measured in the number of edges from s to every other vertex.

In Procedure *bfs*(v) of Algorithm BFS, add the statement $p(w) \leftarrow v$ after w is pushed into the queue, which adds a pointer from w to its parent v in the shortest path tree produced by breadth-first search.

8.17. Use depth-first search to find a spanning tree for the complete bipartite graph $K_{3,3}$. (See Sec. 2.3 for the definition of $K_{3,3}$).

Fig. 8.11(a) shows the complete bipartite graph $K_{3,3}$, and Fig. 8.11(b) shows its depth-first search spanning tree.

8.18. Use breadth-first search to find a spanning tree for the complete bipartite graph $K_{3,3}$. Compare this tree with the tree obtained in Exercise 8.17.

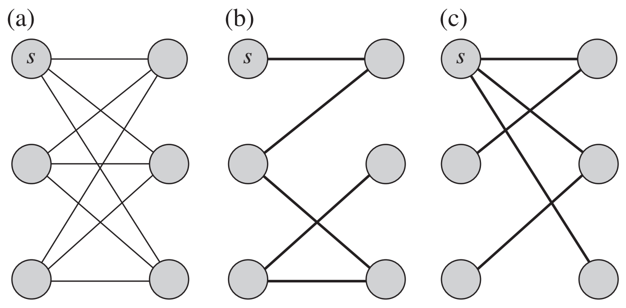

Fig. 8.11 Complete bipartite graph $K_{3,3}$, and its spanning trees.

Fig. 8.11(a) shows the complete bipartite graph $K_{3,3}$, and Fig. 8.11(c) shows its breadth-first search spanning tree. The breadth-first search tree, which is a shortest path tree, has height 2, as opposed to the depth-first search tree whose height is 5.

8.19. Suppose that Algorithm BFS is applied to an undirected graph G. Give an algorithm that classifies the edges of G as either tree edges or cross edges.

In Procedure *bfs*(v) of Algorithm BFS, add the following statement after the test in Line 7:
if w is marked *unvisited* **then output** (v, w) is a tree edge **else output** (v, w) is a cross edge. (See Exercise 8.14).

8.20. Suppose that Algorithm BFS is applied to a directed graph G. Give an algorithm that classifies the edges of G as either tree edges, back edges or cross edges.

Do as in Exercise 8.19, except that edges other than the tree edges are either back or cross edges. To distinguish between back edges and cross edges, use the fact that an edge (v, w) is a back edge if and only if it is not a tree edge, and w is an ancestor of v. (See Exercise 8.15).

8.21. Show that the time complexity of breadth-first search when applied on a graph with n vertices and m edges is $\Theta(n + m)$.

Apply the same analysis as in depth-first search.

8.22. Design an efficient algorithm to determine whether a given graph is bipartite (see Sec. 2.3 for the definition of a bipartite graph).

A graph G is bipartite if and only if its vertices can be colored using two colors. Use breadth-first search to color the vertices of G in a greedy manner: Color the root using color 1, then color the vertices at distance 1 using color 2, etc. until either all vertices have been colored or a conflict is reached.

8.23. Design an algorithm to find a cycle of shortest length in a directed graph. Here, the length of a cycle is measured in terms of its number of edges.

One possibility is to use breadth-first search m times, where m is the number of edges. In each run of Algorithm BFS, delete an edge (u, v) and find the shortest path from v to u. If the length of shortest path found is k, then the length of shortest cycle is $k + 1$. Output the shortest cycle found (see Exercise 8.16).

8.24. Let G be a connected undirected graph, and T the spanning tree resulting from applying breadth-first search on G starting at vertex r. Prove or disprove that the height of T is minimum among all spanning trees with root r.

The height of T is equal to the maximum distance from the root to a leaf node. By the property of shortest-path lengths of breadth-first search, the height of T is minimum among all spanning trees with root r.

9.10 Solutions

9.1. Let Π_1 and Π_2 be two problems such that $\Pi_1 \propto_{poly} \Pi_2$. Suppose that problem Π_2 can be solved in $O(n^k)$ time and the reduction can be done in $O(n^j)$ time. Show that problem Π_1 can be solved in $O(n^{jk})$ time.

Since the reduction can be done in $O(n^j)$ time, the length of the output of the reduction, which is the input to Π_2 is $O(n^j)$. But this implies that applying Π_2 on this input takes $O((n^j)^k) = O(n^{jk})$ steps.

9.2. 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.

We show that the Hamiltonian cycle problem for undirected graphs reduces to the Hamiltonian cycle problem for directed graphs. Given an undirected graph $G = (V, E)$, it is straightforward to convert it to a directed graph $G' = (V, E')$ such that G has a Hamiltonian cycle if and only if G' has a Hamiltonian cycle by replacing each undirected edge in E with two antidirectional edges in the obvious way. Then, there is a Hamiltonian cycle in G if and only if there is a Hamiltonian cycle in G'.

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

We show that PARTITION $\propto_{poly}$ BIN PACKING. Let $S = \{a_1, a_2, \ldots, a_n\}$ be an instance of the problem PARTITION. Let $C = \frac{1}{2}\sum_{i=1}^{n} a_i$. Then, there is a partition of S into S_1 and S_2 such that $\sum_{a_i \in S_1} a_i = \sum_{a_j \in S_2} a_j$ if and only if the items in S can be packed into $k = 2$ bins of size C each. It follows that the problem BIN PACKING is NP-complete, assuming that the problem PARTITION is NP-complete.

9.4. Let Π_1 and Π_2 be two NP-complete problems. Prove or disprove that $\Pi_1 \propto_{poly} \Pi_2$.

By definition of an NP-complete problem, every problem in NP reduces to Π_2 in polynomial time. Since Π_1 is NP-complete, it is in NP, and hence $\Pi_1 \propto_{poly} \Pi_2$.

9.5. 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.

The algorithm tests all possibilities of subgraphs of size k. Since verifying each clique takes $O(n^2)$ time, the overall running time is $O(Kn^2)$, where $K = \binom{n}{k}$. Since, $K = O(n^k)$, the overall running time is $O(n^{k+2})$, which is a polynomial of degree $k+2$. This does not contradict the fact that problem CLIQUE is NP-complete, as k is fixed in this case.

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

Let $G = (V, E)$ be an undirected graph. A subset $S \subseteq V$ is a vertex cover for G if and only if $V - S$ is an independent set in G if and only if $V - S$ is a clique in $\overline{G}$, the complement of G.

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

Let C be a cover for a clique of size n. First, suppose that $|C| \leq n-2$. Then, there are at least two vertices u and v not in the cover. But this implies that the edge (u, v) is not covered. This shows that $|C| \geq n-1$. To show that $n-1$ vertices are sufficient, suppose that vertex x is the only vertex not in the cover. Then, any edge (x, y) is covered. This shows that if C is a cover of minimum size, then $|C| \leq n-1$. It follows that $|C| = n-1$.

9.8. Show that if one can devise a polynomial time algorithm for the problem SATISFIABILITY then NP = P (see Exercise 9.1).

Suppose that there is a polynomial time algorithm for the problem SATISFIABILITY, and let Π be any problem in NP. By definition of NP-complete, $\Pi \propto_{poly}$ SATISFIABILITY. By Exercise 9.1, $\Pi \in$ P.

9.9. In Chapter 6 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.

The problem KNAPSACK can be solved in time $\Theta(nC)$, where C is the knapsack capacity. Hence, the running time is polynomial in input value, but exponential in input size. Thus, there is no contradiction.

9.10. 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).

If linear search is used, the justification may not be valid. Let A be an algorithm to solve the problem TRAVELING SALESMAN. If we call A n times to search for the minimum tour length, the number of calls to Algorithm A is exponential in the input size. Using binary search for the search, however, results in a linear number of calls to Algorithm A measured in the input size.

9.11. Prove that if an NP-complete problem Π is shown to be solvable in polynomial time, then NP = P (see Exercises 9.1 and 9.8).

Suppose that there is a polynomial time algorithm for the problem Π, and let Π' be any problem in NP. By definition of NP-complete, $\Pi' \propto_{poly} \Pi$. By Exercise 9.1, $\Pi' \in$ P.

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

Suppose that NP = P. Then, if Π is NP-complete, we have $\Pi \in$ NP, and thus $\Pi \in$ P. On the other hand, suppose that for some NP-complete problem Π, $\Pi \in$ P. Then, by Exercise 9.11, NP = P.

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

Yes, it is NP-complete. It is easy to see that the problem HAMILTONIAN CYCLE can be reduced to this problem in polynomial time.

9.14. Is the problem LONGEST PATH NP-complete when restricted to directed acyclic graphs? Prove your answer. (See Exercises 9.13 and 6.24).

This problem can be solved in polynomial time (See Exercise 6.24).

9.15. 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.

It is easy to see that the problem LONGEST SIMPLE PATH can be reduced to this problem in polynomial time. The problem LONGEST SIMPLE PATH can be shown to be NP-complete by reducing the problem HAMILTONIAN PATH to it.

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

Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Label the edges so that $E = \{1, 2, 3, \ldots, m\}$. Construct an instance of the problem SET COVER as follows. Let $X = E$, and the set of subsets $\mathcal{F} = \{S_v \mid v \in V\}$, where S_v consists of the set of edges incident to v in G. It is easy to see that there is a vertex cover in G of size k if and only if there is a set cover in the instance of SET COVER of size k.

9.17. Simplify the reduction from the problem SATISFIABILITY to VERTEX COVER by using 3-SAT instead of SATISFIABILITY.

The reduction is the same, except that the cliques will simplify to triangles, and $k = n + \sum_{j=1}^{m}(n_j - 1) = n + \sum_{j=1}^{m} 2 = n + 2m$.

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

The problem TAUTOLOGY, which is complete for the class co-NP, is such that (1) TAUTOLOGY is in P if and only if co-NP = P, and (2) TAUTOLOGY is in NP if and only if co-NP = NP. So, there are more conclusions if it is shown that it is in P or NP. This is to be contrasted with the problem SATISFIABILITY, which is in P if and only if NP = P. This shows that it is likely that problems that are complete for the class co-NP are more difficult than NP-complete problems.

9.19. Prove Theorem 9.8.

Suppose that problem Π and its complement $\overline{\Pi}$ are NP-complete. Since Π is NP-complete, $\overline{\Pi}$ is complete for the class co-NP, and hence any problem Π' in co-NP reduces to $\overline{\Pi}$ in polynomial time. Since $\overline{\Pi}$ is NP-complete, Π' reduces to an NP-complete problem in polynomial time. That is, Π' can be solved by a nondeterministic polynomial time algorithm. In other words, $\Pi' \in$ NP. Since Π' is arbitrary, it follows that co-NP $\subseteq$ NP. Similarly, we can show that NP $\subseteq$ co-NP. That is, co-NP $=$ NP.