SECTION 7.3 Divide-and-Conquer Algorithms and Recurrence Relations

Many of these exercises are fairly straightforward applications of Theorem 2 (or its special case, Theorem 1). The messiness of the algebra and analysis in this section is indicative of what often happens when trying to get reasonably precise estimates for the efficiency of complicated or clever algorithms.

1. Let $f(n)$ be the number of comparisons needed in a binary search of a list of n elements. From Example 1 we know that f satisfies the divide-and-conquer recurrence relation $f(n) = f(n/2) + 2$. Also, 2 comparisons are needed for a list with one element, i.e., $f(1) = 2$ (see Example 3 in Section 3.3 for further discussion). Thus $f(64) = f(32) + 2 = f(16) + 4 = f(8) + 6 = f(4) + 8 = f(2) + 10 = f(1) + 12 = 2 + 12 = 14$.

3. In the notation of Example 4 (all numerals in base 2), we want to multiply $a = 1110$ by $b = 1010$. Note that $n = 2$. Therefore $A_1 = 11$, $A_0 = 10$, $B_1 = 10$ and $B_0 = 10$. We need to form $A_1 - A_0 = 11 - 10 = 01$ and $B_0 - B_1 = 00$. Then we need the following three products: $A_1B_1 = (11)(10)$, $(A_1 - A_0)(B_0 - B_1) = (01)(00)$, and $A_0B_0 = (10)(10)$. In order to from these products, the algorithm would in fact recurse, but let us not worry about that, assumiug instead that we have these answers, namely $A_1B_1 = 0110$, $(A_1 - A_0)(B_0 - B_1) = 0000$, and $A_0B_0 = 0100$. Now we need to shift these products various numbers of places to the left. We shift A_1B_1 $2n = 4$ places and also $n = 2$ places, obtaining 01100000 and 011000; we shift $(A_1 - A_0)(B_0 - B_1)$ $n = 2$ places, obtaining 000000, and we shift A_0B_0 $n = 2$ places and also no places, obtaining 010000 and 0100. Finally we add all five of these binary numbers, obtaining 10001100.

5. This problem is asking us to estimate the number of bit operations needed to do the shifts, additions, and subtractions in multiplying two $2n$-bit integers by the algorithm in Example 4. First recall from Example 8 in Section 3.6 that the number of bit operations needed for an addition of two k-bit numbers is at most $3k$; the same bound holds for subtraction. Let us assume that to shift a number k bits also requires k bit operations. Thus we need to count the number of additions and shifts of various sizes that occur in the fast multiplication algorithm. First, we need to perform two subtractions of n-bit numbers to get $A_1 - A_0$ and $B_0 - B_1$; these will take up to $6n$ bit operations altogether. We need to shift A_1B_1 $2n$ places (requiring $2n$ bit operations), and also n places (requiring n bit operations); we need to shift $(A_1 - A_0)(B_0 - B_1)$ n places (requiring n bit operations); and we need to shift A_0B_0 n places, also requiring n bit operations. This makes a total of $5n$ bit operations for the shifting. Finally we need to worry about the additions (which actually might include a subtraction if the middle term is negative). If we are clever, we can add the four terms that involve at most $3n$ bits first (that is, everything except the $2^{2n}A_1B_1$). Three additions are required, each taking $9n$ bit operations, for a total of $27n$ bit operations. Finally we need to perform one addition involving a $4n$-bit number, taking $12n$ operations. This makes a total of $39n$ bit operations for the additions.

Putting all these operations together, we need perhaps a total of $6n + 5n + 39n = 50n$ bit operations to perform all the additions, subtractions, and shifts. Obviously this bound is not exact; it depends on the actual implementation of these binary operations.

Using $C = 50$ as estimated above, the recurrence relation for fast multiplication is $f(2n) = 3f(n) + 50n$, with $f(1) = 1$ (one multiplication of bits is all that is needed if we have 1-bit numbers). Thus we can compute $f(64)$ as follows: $f(2) = 3 \cdot 1 + 50 = 53$; $f(4) = 3 \cdot 53 + 100 = 259$; $f(8) = 3 \cdot 259 + 200 = 977$; $f(16) = 3 \cdot 977 + 400 = 3331$; $f(32) = 3 \cdot 3331 + 800 = 10793$; and finally $f(64) = 3 \cdot 10793 + 1600 = 33979$. Thus about 34,000 bit operations are needed.

7. We compute these from the bottom up. (In fact, it is easy to see by induction that $f(3^k) = k + 1$, so no computation is really needed at all.)

a) $f(3) = f(1) + 1 = 1 + 1 = 2$ b) $f(9) = f(3) + 1 = 2 + 1 = 3$; $f(27) = f(9) + 1 = 3 + 1 = 4$

c) $f(81) = f(27) + 1 = 4 + 1 = 5$; $f(243) = f(81) + 1 = 5 + 1 = 6$; $f(729) = f(243) + 1 = 6 + 1 = 7$

9. We compute these from the bottom up.

a) $f(5) = f(1) + 3 \cdot 5^2 = 4 + 75 = 79$

b) $f(25) = f(5) + 3 \cdot 25^2 = 79 + 1875 = 1954$; $f(125) = f(25) + 3 \cdot 125^2 = 1954 + 46875 = 48{,}829$

c) $f(625) = f(125) + 3 \cdot 625^2 = 48829 + 1171875 = 1220704$; $f(3125) = f(625) + 3 \cdot 3125^2 = 1220704 + 29296875 = 30{,}517{,}579$

11. We apply Theorem 2, with $a = 1$, $b = 2$, $c = 1$, and $d = 0$. Since $a = b^d$, we have that $f(n)$ is $O(n^d \log n) = O(\log n)$.

13. We apply Theorem 2, with $a = 2$, $b = 3$, $c = 4$, and $d = 0$. Since $a > b^d$, we have that $f(n)$ is $O(n^{\log_b a}) = O(n^{\log_3 2}) \approx O(n^{0.63})$.

15. After 1 round, there are 16 teams left; after 2 rounds, 8 teams; after 3 rounds, 4 teams; after 4 rounds, 2 teams; and after 5 rounds, only 1 team remains, so the tournament is over. In general, k rounds are needed if there are 2^k teams (easily proved by induction).

17. a) Our recursive algorithm will take a sequence of names and determine whether one name occurs as more than half of the elements of the sequence, and if so, which name that is. If the sequence has just one element, then the one person on the list is the winner. For the recursive step, divide the list into two parts—the first half and the second half—as equally as possible. As is pointed out in the hint, no one could have gotten a majority of the votes on this list without having a majority in one half or the other, since if a candidate got less than or equal to half the votes in each half, then he got less than or equal to half the votes in all (this is essentially just the distributive law). Apply the algorithm recursively to each half to come up with at most two names. Then run through the entire list to count the number of occurrences of each of those names to decide which, if either, is the winner. This requires at most $2n$ additional comparisons for a list of length n.

b) We apply the Master Theorem with $a = 2$, $b = 2$, $c = 2$, and $d = 1$. Since $a = b^d$, we know that the number of comparisons is $O(n^d \log n) = O(n \log n)$.

19. a) We compute x^n when n is even by first computing $y = x^{n/2}$ recursively and then doing one multiplication, namely $y \cdot y$. When n is odd, we first compute $y = x^{(n-1)/2}$ recursively and then do two multiplications, namely $y \cdot y \cdot x$. So if $f(n)$ is the number of multiplications required, assuming the worst, then we have essentially $f(n) = f(n/2) + 2$.

b) By the Master Theorem, with $a = 1$, $b = 2$, $c = 2$, and $d = 0$, we see that $f(n)$ is $O(n^0 \log n) = O(\log n)$.

21. a) $f(16) = 2f(4) + 1 = 2(2f(2) + 1) + 1 = 2(2 \cdot 1 + 1) + 1 = 7$

b) Let $m = \log n$, so that $n = 2^m$. Also, let $g(m) = f(2^m)$. Then our recurrence becomes $f(2^m) = 2f(2^{m/2}) + 1$, since $\sqrt{2^m} = (2^m)^{1/2} = 2^{m/2}$. Rewriting this in terms of g we have $g(m) = 2g(m/2) + 1$. Theorem 1 now tells us that $g(m)$ is $O(m^{\log_2 2}) = O(m)$. Since $m = \log n$, this says that our function is $O(\log n)$.

23. a) The messiest part of this is just the bookkeeping. Note that we start with *best* set to 0, since the empty subsequence has a sum of 0, and this is the best we can do if all the terms are negative. Also note that it would be an easy improvement to keep track of where the subsequence is located, as well as what its sum is.

```
procedure largest sum($a_1, \ldots, a_n$ : real numbers)
best := 0 {empty subsequence has sum 0}
for i := 1 to n
begin
        sum := 0
        for j := i to n
        begin
                sum := sum + $a_j$
                if sum > best then best := sum
        end
end
{best is the maximum possible sum of numbers in the list}
```

b) One sum and one comparison are made inside the inner loop. This loop is executed $C(n+1,2)$ times—once for each choice of a pair (i,j) of endpoints of the sequence of consecutive terms being examined (this is a combination with repetition allowed, since $i = j$ when we are examining one term by itself). Since $C(n+1,2) = n(n+1)/2$ the computational complexity is $O(n^2)$.

c) We divide the list into a first half and a second half and apply the algorithm recursively to find the largest sum of consecutive terms for each half. The largest sum of consecutive terms in the entire sequence is either one of these two numbers or the sum of a sequence of consecutive terms that crosses the middle of the list. To find the largest possible sum of a sequence of consecutive terms that crosses the middle of the list, we start at the middle and move forward to find the largest possible sum in the second half of the list, and move backward to find the largest possible sum in the first half of the list; the desired sum is the sum of these two quantities. The final answer is then the largest of this sum and the two answers obtained recursively. The base case is that the largest sum of a sequence of one term is the larger of that number and 0.

d) (i) Split the list into the first half, $-2, 4, -1, 3$, and the second half, $5, -6, 1, 2$. Apply the algorithm recursively to each half (we omit the details of this step) to find that the largest sum in the first half is 6 and the largest sum in the second half is 5. Now find the largest sum of a sequence of consecutive terms that crosses the middle of the list. Moving forward, the best we can do is 5; moving backward, the best we can do is 6. Therefore we can get a sum of 11 by adding the second through fifth terms. This is better than either recursive answer, so the desired answer is 11. (ii) Split the list into the first half, $4, 1, -3, 7$, and the second half, $-1, -5, 3, -2$. Apply the algorithm recursively to each half (we omit the details of this step) to find that the largest sum in the first half is 9 and the largest sum in the second half is 3. Now find the largest snm of a sequence of consecntive terms that crosses the middle of the list. Moving forward, the best we can do is -1; moving backward, the best we can do is 9. Therefore we can get a sum of 8 by crossing the middle. The best of these three possibilities is 9, which we get from the first through fourth terms. (iii) Split the list into the first half, $-1, 6, 3, -4$, and the second half, $-5, 8, -1, 7$. Apply the algorithm recursively to each half (we omit the details of this step) to find that the largest sum in the first half is 9 and the largest sum in the second half is 14. Now find the largest sum of a sequence of consecutive terms that crosses the middle of the list. Moving forward, the best we can do is 9; moving backward, the best we can do is 5. Therefore we can get a sum of 14 by adding the second through eighth terms. The best of these three is actually a tie, between the second through eighth terms and the sixth throngh eighth terms, with a sum of 14 in each case.

e) Let $S(n)$ be the number of sums and $C(n)$ the number of comparisons used. Since the "conquer" step requires n sums and $n+2$ comparisons (two extra comparisons to determine the winner among the three possible largest sums), we have $S(n) = 2S(n/2) + n$ and $C(n) = 2C(n/2) + n + 2$. The basis step is $C(1) = 1$ and $S(1) = 0$.

f) By the Master Theorem with $a = b = 2$ and $d = 1$, we see that we need only $O(n \log n)$ of each type of operation. This is a significant improvement over the $O(n^2)$ complexity we found in part **(b)** for the algorithm in part **(a)**.

25. To carry this down to its base level would require applying the algorithm seven times, so to keep things within reason, we will show only the outermost step. The points are already sorted for us, and so we divide them into two groups, using x coordinate. The left side will have the first eight points listed in it (they all have x coordinates less than 4.5), and the right side will have the rest, all of which have x coordinates greater than 4.5. Thus our vertical line will be taken to be $x = 4.5$. Now assume that we have already applied the algorithm recursively to find the minimum distance between two points on the left, and the minimum distance on the right. It turns out that $d = d_L = d_R = 2$. This is achieved, for example, by tbe points $(1,6)$ and $(3,6)$. Thus we want to concentrate on the strip from $x = 2.5$ to $x = 6.5$ of width $2d$. The only points in this strip are $(3,1)$, $(3,6)$, $(3,10)$, $(4,3)$, $(5,1)$, $(5,5)$, $(5,9)$, and $(6,7)$, Working from the bottom up, we compute distances from these points to points as much as $d = 2$ vertical units above them. According to the discussion in the text, there can never be more than seven such computations for each point in the strip. The distances we need, then, are $\overline{(3,1)(5,1)}$, $\overline{(3,1)(4,3)}$, $\overline{(5,1)(4,3)}$, $\overline{(4,3)(5,5)}$, $\overline{(5,5)(3,6)}$, $\overline{(5,5)(6,7)}$, $\overline{(3,6)(6,7)}$, $\overline{(6,7)(5,9)}$, and $\overline{(5,9)(3,10)}$. The smallest of these turns out to be 2, so the minimum distance $d = 2$ in fact is the smallest distance among all the points. (Actually we did not need to compute the distances between points that were already on the same side of the dividing line, since their distance had already been computed in the recursive step, but checking whether they are on opposite sides of the vertical line would entail additional computation anyway.)

27. The algorithm is essentially the same as the algorithm given in Example 12. The only difference is in constructing the boxes centered on the vertical line that divides the two halves of the set of points. In this variation, our strip still has width $2d$ (i.e., d units to the left and d units to the right of the vertical line), because it would be possible for two points within this box, one on each side of the line, to lie at a distance less than d from each other, but no point outside this strip has a chance to contribute to a small "across the line" distance. In this variation, however, we do not need to construct eight boxes of size $(d/2) \times (d/2)$, but rather just two boxes of size $d \times d$. The reason for this is that there can be at most one point in each of those boxes using the distance formula given in this exercise—two points within such a box (which is on the same side of the dividing line) can be at most d units apart and so would already have been considered in the recursive step. Thus the recurrence relation is the same as the recurrence relation in Example 12, except that the coefficient 7 is replaced by 1. The analysis via the Master Theorem remains the same, and again we get a $O(n \log n)$ algorithm.

29. Suppose $n = b^k$, so that $k = \log_b n$. We will prove by induction on k that $f(b^k) = f(1)(b^k)^d + c(b^k)^d k$, which is what we are asked to prove, translated into this notation. If $k = 0$, then the equation reduces to $f(1) = f(1)$, which is certainly true. We assume the inductive hypothesis, that $f(b^k) = f(1)(b^k)^d + c(b^k)^d k$, and we try to prove that $f(b^{k+1}) = f(1)(b^{k+1})^d + c(b^{k+1})^d(k+1)$. By the recurrence relation for $f(n)$ in terms of $f(n/b)$, we have $f(b^{k+1}) = b^d f(b^k) + c(b^{k+1})^d$. Then we invoke the inductive hypothesis and work through the algebra:

$$\begin{aligned} b^d f(b^k) + c(b^{k+1})^d &= b^d\big(f(1)(b^k)^d + c(b^k)^d k\big) + c(b^{k+1})^d \\ &= f(1)b^{kd+d} + cb^{kd+d}k + c(b^{k+1})^d \\ &= f(1)(b^{k+1})^d + c(b^{k+1})^d k + c(b^{k+1})^d \\ &= f(1)(b^{k+1})^d + c(b^{k+1})^d(k+1) \end{aligned}$$

31. The algebra is quite messy, but this is a straightforward proof by induction on $k = \log_b n$. If $k = 0$, so that $n = 1$, then we have the true statement

$$f(1) = C_1 + C_2 = \frac{b^d c}{b^d - a} + f(1) + \frac{b^d c}{a - b^d}$$

(since the fractions cancel each other out). Assume the inductive hypothesis, that for $n = b^k$ we have

$$f(n) = \frac{b^d c}{b^d - a} n^d + \left(f(1) + \frac{b^d c}{a - b^d} \right) n^{\log_b a} .$$

Then for $n = b^{k+1}$ we apply first the recurrence relation, then the inductive hypothesis, and finally some algebra:

$$\begin{aligned} f(n) &= af\left(\frac{n}{b}\right) + cn^d \\ &= a\left(\frac{b^d c}{b^d - a}\left(\frac{n}{b}\right)^d + \left(f(1) + \frac{b^d c}{a - b^d} \right)\left(\frac{n}{b}\right)^{\log_b a} \right) + cn^d \\ &= \frac{b^d c}{b^d - a} \cdot n^d \cdot \frac{a}{b^d} + \left(f(1) + \frac{b^d c}{a - b^d} \right) n^{\log_b a} + cn^d \\ &= n^d \left(\frac{ac}{b^d - a} + \frac{c(b^d - a)}{b^d - a} \right) + \left(f(1) + \frac{b^d c}{a - b^d} \right) n^{\log_b a} \\ &= \frac{b^d c}{b^d - a} \cdot n^d + \left(f(1) + \frac{b^d c}{a - b^d} \right) n^{\log_b a} \end{aligned}$$

Thus we have verified that the equation holds for $k + 1$, and our induction proof is complete.

33. The equation given in Exercise 31 says that $f(n)$ is the sum of a constant times n^d and a constant times $n^{\log_b a}$. Therefore we need to determine which term dominates, i.e., whether d or $\log_b a$ is larger. But we are given $a > b^d$; hence $\log_b a > \log_b b^d = d$. It therefore follows (we are also using the fact that f is increasing) that $f(n)$ is $O(n^{\log_b a})$.

35. We use the result of Exercise 33, since $a = 5 > 4^1 = b^d$. Therefore $f(n)$ is $O(n^{\log_b a}) = O(n^{\log_4 5}) \approx O(n^{1.16})$.

37. We use the result of Exercise 33, since $a = 8 > 2^2 = b^d$. Therefore $f(n)$ is $O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$.