SECTION 3.3 Complexity of Algorithms

Some of these exercises involve analysis of algorithms, as was done in the examples in this section. These are a matter of carefully counting the operations of interest, usually in the worst case. Some of the others are algebra exercises that display the results of the analysis in real terms--the number of years of computer time, for example, required to solve a large problem. **Horner's method** *for evaluating a polynomial, given in Exercise 8, is a nice trick to know. It is extremely handy for polynomial evaluation on a pocket calculator (especially if the calculator is so cheap that it does not use the usual precedence rules).*

use one comparison for the final assignment of *location*. Thus $2i + 1$ comparisons are needed. We need to average the numbers $2i + 1$ (for i from 1 to n), to find the average number of comparisons needed for a successful search. This is $(3 + 5 + \cdots + (2n + 1))/n = (n + (2 + 4 + \cdots + 2n))/n = (n + 2(1 + 2 + \cdots + n))/n = (n + 2(n(n+1)/2))/n = n + 2$. Finally, we average the $2n+2$ comparisons for the unsuccessful search with this average $n + 2$ comparisons for a successful search to obtain a grand average of $(2n + 2 + n + 2)/2 = (3n + 4)/2$ comparisons.

15. We will count comparisons of elements in the list to x. (This ignores comparisons of subscripts, but since we are only interested in a big-O analysis, no harm is done.) Furthermore, we will assume that the number of elements in the list is a power of 3, say $n = 3^k$. Just as in the case of binary search, we need to determine the maximum number of times the **while** loop is iterated. Each pass through the loop cuts the number of elements still being considered (those whose subscripts are from i to j) by a factor of 3. Therefore after k iterations, the active portion of the list will have length 1; that is, we will have $i = j$. The loop terminates at this point. Now each iteration of the loop requires two comparisons in the worst case (one with a_u and one with a_l). Two more comparisons are needed at the end. Therefore the number of comparisons is $2k+2$, which is $O(k)$. But $k = \log_3 n$, which is $O(\log n)$ since logarithms to different bases differ only by multiplicative constants, so the time complexity of this algorithm (in all cases, not just the worst case) is $O(\log n)$.

17. The algorithm we gave for finding a mode essentially just goes through the list once, doing a little bookkeeping at each step. In particular, between any two successive executions of the statement $i := i + 1$ there are at most about six operations (such as comparing *count* with *modecount*, or reinitializing *value*). Therefore at most about $6n$ steps are done in all, so the time complexity in all cases is $O(n)$.

19. The worst case is that in which we do not find any term equal to some previous term. In that case, we need to go through all the terms a_2 through a_n, and for each of those, we need to go through all the terms occurring prior to that term. Thus the inner loop of our algorithm is executed once for $i = 2$ (namely for $j = 1$), twice for $i = 3$ (namely for $j = 1$ and $j = 2$), three times for $i = 4$, and so on, up to $n - 1$ times for $i = n$. Thus the number of comparisons that need to be made in the inner loop is $1 + 2 + 3 + \cdots + (n - 1)$. As was mentioned in this section (and will be shown in Section 4.1), that sum is $(n - 1)(n - 1 + 1)/2$, which is clearly $O(n^2)$ but no better. Bookkeeping details do not increase this estimate.

21. We needed to go through the sequence only once, making one comparison of terms (and two bookkeeping comparisons) until we found the desired term (or had exhausted the list). Thus this algorithm's time complexity is clearly $O(n)$.

23. We had to read the string (or at least half of it) simultaneously from the front and the back and compare characters to make sure they were equal. Thus we will need at least $\lfloor n/2 \rfloor$ comparisons in the worst case, which is $O(n)$.

25. Since we are doing binary search to find the correct location of the j^{th} element among the first $j - 1$ elements, which are already sorted, we need $O(\log n)$ comparisons for each element. Therefore we use $O(n \log n)$ comparisons in all. The cost of swapping of items to make room for the insertions is $O(n^2)$, however (the originally first item may need to move $n - 1$ places, the second item $n - 2$ places, and so on).

27. **a)** The linear search algorithm uses about n comparisons for a list of length n, and $2n$ comparisons for a list of length $2n$. Therefore the number of comparisons, like the size of the list, doubles.

b) The binary search algorithm uses about $\log n$ comparisons for a list of length n, and $\log(2n) = \log 2 + \log n = 1 + \log n$ comparisons for a list of length $2n$. Therefore the number of comparisons increases by about 1.

1. Assuming that the algorithm given to find the smallest element of a list is identical to Algorithm 1 in Section 3.1, except that the inequality is reversed (and the name *max* replaced by the name *min*), the analysis will be identical to the analysis given in Example 1 in the current section. In particular, there will be $2n - 1$ comparisons needed, counting the bookkeeping for the loop.

3. The linear search would find this element after at most 9 comparisons (4 to determine that we have not yet finished with the **while** loop, 4 more to determine if we have located the desired element yet, and 1 to set the value of *location*). Binary search, according to Example 3, will take $2 \log 32 + 2 = 2 \cdot 5 + 2 = 12$ comparisons. Since $9 < 12$, the linear search will be faster, in terms of comparisons.

5. The algorithm simply scans the bits one at a time. Thus clearly $O(n)$ comparisons are required (perhaps one for bookkeeping and one for looking at the i^{th} bit, for each i from 1 to n).

7. **a)** Here we have $n = 2$, $a_0 = 1$, $a_1 = 1$, $a_2 = 3$, and $c = 2$. Initially, we set *power* equal to 1 and y equal to 1. The first time through the **for** loop (with $i = 1$), *power* becomes 2 and so y becomes $1 + 1 \cdot 2 = 3$. The second and final time through the loop, *power* becomes $2 \cdot 2 = 4$ and y becomes $3 + 3 \cdot 4 = 15$. Thus the value of the polynomial at $x = 2$ is 15.

b) Each pass through the loop requires two multiplications and one addition. Therefore there are a total of $2n$ multiplications and n additions in all.

9. This is an exercise in algebra, numerical analysis (for some of the parts), and using a calculator. Since each bit operation requires 10^{-9} seconds, we want to know for what value of n there will be at most 10^9 bit operations required. Thus we need to set the expression equal to 10^9, solve for n, and round down if necessary.

a) Solving $\log n = 10^9$, we get (recalling that "log" means logarithm base 2) $n = 2^{10^9}$. By taking $\log_{10}$ of both sides, we find that this number is approximately equal to $10^{300,000,000}$. Obviously we do not want to write out the answer explicitly!

b) Clearly $n = 10^9$.

c) Solving $n \log n = 10^9$ is not trivial. There is no good formula for solving such **transcendental** equations. An algorithm that works well with a calculator is to rewrite the equation as $n = 10^9 / \log n$, enter a random starting value, say $n = 2$, and repeatedly calculate a new value of n. Thus we would obtain, in succession, $n = 10^9 / \log 2 = 10^9$, $n = 10^9 / \log(10^9) \approx 33{,}447{,}777.3$, $n = 10^9 / \log(33{,}447{,}777.3) \approx 40{,}007{,}350.14$, $n = 10^9 / \log(40{,}007{,}350.14) \approx 39{,}598{,}061.08$, and so on. After a few more iterations, the numbers stabilize at approximately 39,620,077.73, so the answer is 39,620,077.

d) Solving $n^2 = 10^9$ gives $n = 10^{4.5}$, which is 31,622 when rounded down.

e) Solving $2^n = 10^9$ gives $n = \log(10^9) \approx 29.9$. Rounding down gives the answer, 29.

f) The quickest way to find the largest value of n such that $n! \leq 10^9$ is simply to try a few values of n. We find that $12! \approx 4.8 \times 10^8$ while $13! \approx 6.2 \times 10^9$, so the answer is 12.

11. In each case, we just multiply the number of seconds per operation by the number of operations (namely 2^{50}). To convert seconds to minutes, we divide by 60; to convert minutes to hours, we divide by 60 again. To convert hours to days, we divide by 24; to convert days to years, we divide by $365\frac{1}{4}$.

a) $2^{50} \times 10^{-6} = 1{,}125{,}899.907$ seconds ≈ 36 years

b) $2^{50} \times 10^{-9} = 1{,}125{,}899.907$ seconds ≈ 13 days

c) $2^{50} \times 10^{-12} = 1{,}125.899907$ seconds ≈ 19 minutes

13. If the element is not in the list, then $2n + 2$ comparisons are needed: two for each pass through the loop, one more to get out of the loop, and one more for the statement just after the loop. If the element is in the list, say as the i^{th} element, then we need to enter the loop i times, each time costing two comparisons, and