SECTION 3.3 Complexity of Algorithms

2. We can sort the first four elements (which we assume are distinct integers) by the comparisons shown. (After the first three steps, the smallest element is definitely in the first position; after the next two steps, the next smallest element is in the second position.) Since only six comparisons are used, regardless of the length of the list, this algorithm has complexity $O(1)$.

procedure *sort four*$(a_1, a_2, \ldots, a_n$: distinct integers)
if $a_4 < a_3$ **then** interchange a_4 and a_3
if $a_3 < a_2$ **then** interchange a_3 and a_2
if $a_2 < a_1$ **then** interchange a_2 and a_1
if $a_4 < a_3$ **then** interchange a_4 and a_3
if $a_3 < a_2$ **then** interchange a_3 and a_2
if $a_4 < a_3$ **then** interchange a_4 and a_3

4. If we successively square k times, then we have computed x^{2^k}. Thus we can compute x^{2^k} with only k multiplications, rather than the $2^k - 1$ multiplications that the naive algorithm would require, so this method is much more efficient.

6. **a)** By the way that $S - 1$ is defined, it is clear that $S \wedge (S - 1)$ is the same as S except that the rightmost 1 bit has been changed to a 0. Thus we add 1 to *count* for every one bit (since we stop as soon as $S = 0$, i.e., as soon as S consists of just 0 bits).

b) Obviously the number of bitwise *AND* operations is equal to the final value of *count*, i.e., the number of one bits in S.

8. **a)** Initially $y := 3$. For $i = 1$ we set y to $3 \cdot 2 + 1 = 7$. For $i = 2$ we set y to $7 \cdot 2 + 1 = 15$, and we are done.

b) There is one multiplication and one addition for each of the n passes through the loop, so there are n multiplications and n additions in all.

10. We are asked to compute $(2n^2 + 2^n) \cdot 10^{-9}$ for each of these values of n. When appropriate, we change the units from seconds to some larger unit of time.

a) 1.224×10^{-6} seconds **b)** approximately 1.05×10^{-3} seconds

c) approximately 1.13×10^{6} seconds, which is about 13 days (nonstop)

d) approximately 1.27×10^{21} seconds, which is about 4×10^{13} years (nonstop)

12. **a)** The number of comparisons does not depend on the values of a_1 through a_n. Exactly $2n - 1$ comparisons are used, as was determined in Example 1. In other words, the best case performance is $O(n)$.

b) In the best case $x = a_1$. We saw in Example 4 that three comparisons are used in that case. The best case performance, then, is $O(1)$.

c) It is hard to give an exact answer, since it depends on the binary representation of the number n, among other things. In any case, the best case performance is really not much different from the worst case performance, namely $O(\log n)$, since the list is essentially cut in half at each iteration, and the algorithm does not stop until the list has only one element left in it.

14. **a)** In order to find the maximum element of a list of n elements, we need to make at least $n - 1$ comparisons, one to rule out each of the other elements. Since Algorithm 1 in Section 3.1 used just this number (not counting bookkeeping), it is optimal.

b) Linear search is not optimal, since we found that binary search was more efficient. This assumes that we can be given the list already sorted into increasing order.

16. 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 4, say $n = 4^k$. 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 4. 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_m and one with either a_l or a_u). Three more comparisons are needed at the end. Therefore the number of comparisons is $2k + 3$, which is $O(k)$. But $k = \log_4 n$, 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 $O(\log n)$.

18. The algorithm we gave for finding all the modes 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 eight operations (such as comparing *count* with *modecount*, or reinitializing *value*). Therefore at most about $8n$ steps are done in all, so the time complexity in all cases is $O(n)$.

20. The algorithm we gave is clearly of linear time complexity, i.e., $O(n)$, since we were able to keep updating the sum of previous terms, rather than recomputing it each time. This applies in all cases, not just the worst case.

22. The algorithm read through the list once and did a bounded amount of work on each term. Looked at another way, only a bounded amount of work was done between increments of j in the algorithm given in the solution. Thus the complexity is $O(n)$.

24. It takes $n-1$ comparisons to find the least element in the list, then $n-2$ comparisons to find the least element among the remaining elements, and so on. Thus the total number of comparisons is $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$, which is $O(n^2)$.

26. Each iteration (determining whether we can use a coin of a given denomination) takes a bounded amount of time, and there are at most n iterations, since each iteration decreases the number of cents remaining. Therefore there are $O(n)$ comparisons.

28. a) The bubble sort algorithm uses about $n^2/2$ comparisons for a list of length n, and $(2n)^2/2 = 2n^2$ comparisons for a list of length $2n$. Therefore the number of comparisons goes up by a factor of 4.

b) The analysis is the same as for bubble sort.

c) The analysis is the same as for bubble sort.

d) The binary insertion sort algorithm uses about $Cn \log n$ comparisons for a list of length n, where C is a constant. Therefore it uses about $C \cdot 2n \log 2n = C \cdot 2n \log 2 + C \cdot 2n \log n = C \cdot 2n + C \cdot 2n \log n$ comparisons for a list of length $2n$. Therefore the number of comparisons increases by about a factor of 2 (for large n, the first term is small compared to the second and can be ignored).