memory. If we have only a fixed number k of processors, then previous complexity bounds are changed by at most $1/k$. If we allow the number of processors to grow as the size of the data grows, however, then problem complexity is sometimes reduced significantly.

At the time of this writing, development of quantum computers, which would depend on quantum-mechanical principles to implement massive amounts of parallelism, look to be conceivable. For such computers, complexity analysis might also change.

5.4 Exercises

1. Calculate how many times statement S is executed in each block of code. Simplify all your answers.

(a)
```
I = 1
while I ≤ N
    S
    I = 2 · I
```

(b)
```
Counter1 = 1
while Counter1 ≤ N − 1
    Counter2 = 1
    while Counter2 ≤ N − 1
        S
        Counter2 = Counter2 + 1
    Counter1 = Counter1 + 1
```

(c)
```
Counter1 = 1
while Counter1 < X
    Counter2 = 1
    while Counter2 ≤ X
        S
        Counter2 = Counter2 + 2
    Counter1 = Counter1 + 1
```

(d)
```
Counter1 = 1
while Counter1 ≤ N
    Counter2 = 1
    while Counter2 ≤ N
        S
        Counter2 = Counter2 + 1
    Counter3 = 1
    while Counter3 ≤ N
        Counter4 = 1
        while Counter4 ≤ N
            S
            Counter4 = Counter4 + 1
        Counter3 = Counter3 + 1
    Counter1 = Counter1 + 1
```

(e) *Counter1* = 1
while *Counter1* ≤ *X*
Counter2 = 1
while *Counter2* ≤ *X*
S
Counter2 = 2 ∗ *Counter2*
Counter1 = *Counter1* + 1

(f) *Counter1* = 1
while *Counter1* ≤ *N* − 1
Counter2 = 1
while *Counter2* ≤ *Counter1*
S
Counter2 = *Counter2* + 1
Counter1 = *Counter1* + 1

2. Confirm the claims of parts (a) and (b) of Example 1.
3. Show that the following SelectionSort algorithm, on any input list of size n, makes exactly $n(n-1)/2$ comparisons and exactly $n-1$ calls to *Swap*.

Algorithm: SelectionSort

INPUT: A list of distinct values *List*[1], . . . , *List*[*N*]
OUTPUT: *List*[1], . . . , *List*[*N*] with values in increasing order

SelectionSort(*List*, *N*)
for *Position* = 1 to *N* − 1 do
Small = *List*[*Position*]
Place = *Position*
for *I* = *Position* + 1 to *N* do
if (*Small* > *List*[*I*]) then
Small = *List*[*I*]
Place = *I*
Swap(*List*[*Position*], *List*[*Place*])

4. Suppose you record the run times of program P. You find out that on inputs of length 1, its run time is always 1/8 second, and on inputs of length 10, its run time is always 1/4 second. Under each of the following assumptions, calculate how long it will take to run the program on inputs of length 20, 100, 1000, and 10,000:
   (a) For some constants a and b (the values of which you must determine), running the program on a data set of size d always takes $ad + b$ seconds.
   (b) For some constants a and b, running the program on a data set of size d always takes $ad^2 + b$ seconds.

(c) For some constants a and b, running the program on a data set of size d always takes $a \cdot 2^{bd}$ seconds.

5. Write an algorithm in pseudocode that follows the description in Example 2 for the special case where the formula ϕ is in CNF. For the sake of simplicity, assume that each proposition letter is a lowercase letter (a–z), but try to ignore the fact that there are only 26 such letters. Also, assume that each of the symbols ($\wedge$, $\vee$, $\neg$) is also just one character, some ASCII character other than a through z. (*Hint:* Suppose you are given the formula from that example,

$$\phi = (a \vee b \vee c \vee d \vee \neg b) \wedge (c \vee d \vee \neg c \vee e \vee f)$$

and the example truth assignment of *TRUE* to a, $\neg b$, c, $\neg d$, e, and $\neg f$. Your program should scan through ϕ from left to right, keeping track of (i) the value *so far* of the current clause—the disjunction of all the literals seen so far in that clause—and (ii) the value *so far* of the entire formula—the conjunction of the values of all the clauses completed so far.)

Since this can be done with just one scan through the formula, you may be tempted to think the algorithm is linear time, but it is not. Why? For formulas not in CNF, the algorithm is also $\mathbf{O}(n^2)$ but more complicated: One must first *parse* the formula to get an expression tree and then keep track of truth values on the tree. The student may have seen such problems in a data structures or algorithms course.

6. For each of the problems (a)–(d) below;
   (i) Write an algorithm in pseudocode to solve the problem (be sure your algorithm works correctly if $m = 0$ or $n = 0$; it should not make any assignments to elements of the array), and
   (ii) Calculate how many assignment statements and how many comparisons the algorithm causes to be executed as a function of m and n. In this case, count assignments to and comparisons of *index variables,* as well as assignments to and comparisons of positions in the array. Simplify your answers.

   (a) Initialize all the elements of an $m \times n$ array to 0.
   (b) Initialize all the elements of an $m \times n$ array that lie on or above the diagonal to 0. (Here, by "diagonal" we mean positions $[r, c]$ where $r = c$.)
   (c) Initialize all the elements of an $m \times n$ array that lie above the diagonal to 0.
   (d) Initialize all the elements on the diagonal of an $m \times n$ array on the diagonal to 0.

7. It is tempting to compute the complexity of an algorithm by counting statements, as we did with the *BubbleSort* example, but only keeping track of the number of steps along the way up to $\mathbf{O}(*)$. This turns out not to work with loops. For example, it is possible that each time through the loop, the number of statements executed is in $\mathbf{O}(1)$, but that the number of statements executed by a loop of length n is not in $\mathbf{O}(n)$. Find an example. (*Hint:* Each time through the loop, the number of statements executed may be in $\mathbf{O}(1)$, but the constants c and N_0 may change).
8. Compare the graphs of $F_1(x) = 3 \ln(x + 1)$, $F_2(x) = 2x$, $F_3(x) = x^2$, $F_4(x) = x^3$, $F_5(x) = 2^{x-1}$, and $F_6(x) = 3^{x-1}$. What does this suggest about the usefulness of nonpolynomial-time algorithms?
9. Is it reasonable to consider all polynomial-time algorithms to be practical? Why, or why not?