SECTION 3.3 REVIEW

TECHNIQUE

- Do a worst-case analysis of an algorithm either directly from the algorithm description or from a recurrence relation.

MAIN IDEAS

- Analysis of an algorithm estimates the number of basic operations that the algorithm performs, which is dependent on the size of the input.
- Analysis of recursive algorithms often leads to recurrence relations.
- Lacking an exact expression for the number of operations an algorithm performs, it may be possible to find an upper bound.

EXERCISES 3.3

1. Modify the algorithm of Example 27 so that in addition to dropping the student's lowest quiz grade, the highest quiz grade is counted twice (like the old version, your new algorithm should do no operations besides addition and subtraction).
2. What is the total number of arithmetic operations done in the algorithm of Exercise 1?
3. The following algorithm adds all the entries in a square $n \times n$ array A. Analyze this algorithm where the work unit is the addition operation.

```
sum = 0
for i = 1 to n do
    for j = 1 to n do
        sum = sum + A[i, j]
    end for
end for
write ("Total of all array elements is", sum)
```

4. The following algorithm adds all the entries in the "upper triangular" part of a square $n \times n$ array A. Analyze this algorithm where the work unit is the addition operation.

```
sum = 0
for k = 1 to n do
    for j = k to n do
        sum = sum + A[k, j]
    end for
end for
write ("Total of all upper triangular array elements is", sum)
```

5. Analyze the following algorithm where the work unit is the output statement. Assume that $n = 2^m$ for some positive integer m.

```
integer j, k
for k = 1 to n do
    j = n;
    while j ≥ 2 do
        write j
        j = j/2
    end while
end for
```

6. Analyze the following algorithm where the work unit is the output statement. (*Hint:* One of the exercises in Section 2.2 might be helpful).

```
integer i
real d, x;
for i = 1 to n do
   d = 1.0/i;
   x = i;
   while x > 0 do
      write x
      x = x − d;
   end while
end for
```

Exercises 7 and 8 involve $n! = n(n-1)(n-2)\cdots 1$.

7. a. Write the body of an iterative function to compute $n!$ for $n \geq 1$.
   b. Analyze this function where the work unit is the multiplication operation.

8. a. Write a recursive function to compute $n!$ for $n \geq 1$.
   b. Write a recurrence relation for the work done by this function where multiplication is the unit of work.
   c. Solve the recurrence relation of part b.
   d. Compare your answer in part c to your result in Exercise 7b.

Exercises 9 and 10 involve evaluating a polynomial $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_0$ for a specific value of x.

9. A straightforward algorithm to evaluate a polynomial is given by the following function:

```
Poly(real $a_n$, real $a_{n-1}$, ... , real $a_0$, real c, integer n)
//evaluates polynomial $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_0$     for $x = c$
Local variables:
integer i
real sum = $a_0$
real product = 1

   for i = 1 to n do
      product = product * c
      sum = sum + $a_i$ * product
   end for
   return sum
end function Poly
```

a. Walk through this algorithm to compute the value of $2x^3 - 7x^2 + 5x - 14$ for $x = 4$.
b. The algorithm involves both additions and multiplications; analyze this algorithm where those operations are the work units.

10. An alternative to the polynomial evaluation algorithm in Exercise 9 is an algorithm called *Horner's method*. Horner's method relies on an alternative expression for a polynomial, for example

$$2x^3 - 7x^2 + 5x - 14 = -14 + x(5 + x(-7 + x(2)))$$

```
Horner(real $a_n$, real $a_{n-1}$, … , real $a_0$, real $c$, integer $n$)
//evaluates polynomial $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_0$    for $x = c$
//using Horner's method
Local variables:
integer i
real $result = a_n$

   for i = 1 to n do
      $result = result * c + a_{n-i}$
   end for
   return result
end function Horner
```

a. Walk through this algorithm to compute the value of $2x^3 - 7x^2 + 5x - 14$ for $x = 4$.

b. Analyze this algorithm where addition and multiplication operations are the work units.

c. In evaluating a polynomial of degree $n = 98$ for some value of x, how many operations have been saved by using Horner's method over the method of Exercise 9?

11. For the algorithm of Example 27, count the total number of assignments and comparisons done in the best case (least work) and the worst case (most work); describe each of these cases.

12. a. Write a function to convert a binary string $b_nb_{n-1} \dots b_1b_0$ to its decimal equivalent.
    b. Test your function on the binary string 10011
    c. Describe the worst case for this algorithm and find the number of multiplications and additions done in this case.
    d. Describe the best case for this algorithm and find the number of multiplications and additions done in this case.

Exercises 13 and 14 relate to a recursive sorting algorithm called *BubbleSort*.

13. Algorithm *BubbleSort* works by making repeated passes through a list; on each pass, adjacent elements that are out of order are exchanged. At the end of pass 1, the maximum element has "bubbled up" to the end of the list and does not participate in subsequent passes. The following algorithm is called initially with $j = n$.

```
BubbleSort(list L; integer j)
//recursively sorts the items from 1 to j in list L into increasing order

   if $j = 1$ then
      sort is complete, write out the sorted list
   else
      for $i = 1$ to $j - 1$ do
         if $L[i] > L[i + 1]$ then
            exchange $L[i]$ and $L[i + 1]$
         end if
      end for
      BubbleSort($L, j - 1$)
   end if
end function BubbleSort
```

a. Walk through algorithm *BubbleSort* to sort the list 5, 6, 3, 4, 8, 2.

b. Write a recurrence relation for the number of comparisons of list elements done by this algorithm to sort an n-element list.

c. Solve this recurrence relation.

14. In algorithm *BubbleSort*, suppose we include exchanges of list elements as a work unit, in addition to comparisons between list elements.

a. Describe the worst case and find the number of comparisons and exchanges done in this case.

b. Describe the best case and find the number of comparisons and exchanges done in this case.

c. Assume that on the average exchanges between elements must be done about half the time. Find the number of comparisons and exchanges done in this case.

Exercises 15–18 refer to the recursive algorithm *SelectionSort* of Section 3.1.

15. In one part of algorithm *SelectionSort*, the index of the maximum item in a list must be found. This requires comparisons between list elements. In an n-element (unsorted) list, how many such comparisons are needed in the worst case to find the maximum element? How many such comparisons are needed in the average case?

16. Defining the basic operation as the comparison of list elements and ignoring the amount of work required to exchange list elements, write a recurrence relation for the amount of work done by selection sort on an n-element list. (*Hint:* Use the result from Exercise 15.)

17. Solve the recurrence relation of Exercise 16.

18. Assume that the exchange of $L[i]$ and $L[j]$ takes place even if $i = j$. Write an expression for the total number of comparisons and exchanges done to sort an n-element list.

Exercises 19–24 relate to a recursive sorting algorithm called *MergeSort*, which is described as follows: A one-element list is already sorted; no further work is required. Otherwise, split the list in half, sort each half using *MergeSort* (this is the recursive part), and then merge the two halves back into one sorted list.

19. The merge part of algorithm *MergeSort* requires comparing elements from each of two sorted lists to see which goes next into the combined, sorted list. When one list runs out of elements, the remaining elements from the other list can be added without further comparisons. Given the following pairs of lists, perform a merge and count the number of comparisons to merge the two lists into one.

a. 6, 8, 9 and 1, 4, 5

b. 1, 5, 8 and 2, 3, 4

c. 0, 2, 3, 4, 7, 10 and 1, 8, 9

20. Under what circumstances will the maximum number of comparisons take place while merging two sorted lists? If the lengths of the lists are r and s, what is the maximum number of comparisons?

21. Write a recurrence relation for the number of comparisons between list elements done by algorithm *MergeSort* in the worst case. Assume that $n = 2^m$.

22. Solve the recurrence relation of Exercise 21.

23. Use the results of Exercises 18 and 22 to compare the worst-case behavior of *SelectionSort* (counting comparisons and exchanges) and *MergeSort* (counting comparisons) for $n = 4$, 8, 16, and 32 (use a calculator or spreadsheet).

24. Use the results of Exercises 14 and 22 to compare the worst-case behavior of *BubbleSort* (counting comparisons and exchanges) and *MergeSort* (counting comparisons) for $n = 4$, 8, 16, and 32 (use a calculator or spreadsheet).

Exercises 25–34 relate to a recursive sorting algorithm called *QuickSort*, which is described as follows: A one-element list is already sorted; no further work is required. Otherwise, take the first element in the list, call it the pivot element, then walk through the original list to create two new sublists, L_1 and L_2. L_1 consists of all elements that are less than the pivot element and L_2 consists of all elements that are greater than the pivot element. Put the pivot element between L_1 and L_2. Sort each of $L1$ and $L2$ using *QuickSort* (this is the recursive part). Eventually all lists will consist of 1 element sublists separated by previous pivot elements, and at this point the entire original list is in sorted order. This is a little confusing, so here is an example, where pivot elements are shown in brackets:

Original list: 6, 2, 1, 7, 9, 4, 8
After 1st pass: 2, 1, 4, [6], 7, 9, 8
After 2nd pass: 1, [2], 4, [6], [7], 9, 8
After 3rd pass: 1, [2], 4, [6], [7], 8, [9] Sorted

25. Illustrate *QuickSort* as above using the list 9, 8, 3, 13.
26. Illustrate *QuickSort* as above using the list 8, 4, 10, 5, 9, 6, 14, 3, 1, 12, 11.
27. How many comparisons between list elements are required for pass 1 of *QuickSort* in the example list?
28. How many comparisons between list elements are required for pass 1 of *QuickSort* on an n-element list?
29. Suppose that for each pass, each pivot element splits its sublist into two equal-length lists, each approximately half the size of the sublist (which is actually very difficult to achieve). Write a recurrence relation for the number of comparisons between list elements in this case.
30. Solve the recurrence relation of Exercise 29.
31. Suppose that for each pass, each pivot element splits its sublist (which has k elements) into one empty list and one list of size $k - 1$. Write a recurrence relation for the number of comparisons between list elements in this case.
32. Solve the recurrence relation of Exercise 31.
33. Unlike the situation described in Exercise 29 where each pivot element splits the sublist in half for the next pass, the situation described in Exercise 31 can easily occur. Describe a characteristic of the original list that would cause this to happen.
34. Exercise 29 describes the best case of *QuickSort* and Exercise 31 describes the worst case of *QuickSort* with respect to comparisons between list elements.
    a. To which sorting algorithm (*SelectionSort*, *BubbleSort*, *MergeSort*) is the best case of *QuickSort* comparable in the number of comparisons required?
    b. To which sorting algorithm (*SelectionSort*, *BubbleSort*, *MergeSort*) is the worst case of *QuickSort* comparable in the number of comparisons required?

Exercises 35 and 36 refer to algorithm *SequentialSearch*. It is not hard to do an average case analysis of the sequential search algorithm under certain assumptions. Given an n-element list and a target value x for which we are searching, the basic operation is a comparison of list elements to x, hence an analysis should count how many times such an operation is performed "on the average." The definition of "average" is shaped by our assumptions.

35. Assume that x is in the list and is equally to be found at any of the n positions in the list. Fill in the rest of the table giving the number of comparisons for each case.

Position at Which x Occurs	Number of Comparisons
1	1
2	
3	
⋮	
n	

Find the average number of comparisons by adding the results from the table and dividing by n. (*Hint*: See Practice 7 of Section 2.2—we told you that you should remember this!)

36. Find the average number of comparisons under the assumption that x is equally likely to be at any of the n positions in the list or not in the list.

Exercises 37–40 concern a better upper bound for the number of divisions required by the Euclidean algorithm in finding gcd(a, b). Assume that a and b are positive integers with $a > b$.

37. Suppose that m divisions are required to find gcd(a, b). Prove by induction that for $m \geq 1$, it is true that $a \geq F(m + 2)$ and $b \geq F(m + 1)$, where $F(n)$ is the Fibonacci sequence. (*Hint:* To find gcd(a, b), after the first division the algorithm computes gcd(b, r).)
38. Suppose that m divisions are required to find gcd(a, b), with $m \geq 4$. Prove that

$$\left(\frac{3}{2}\right)^{m+1} < F(m + 2) \leq a$$

(*Hint:* Use the result of Exercise 37 here and Exercise 26 of Section 3.1.)
39. Suppose that m divisions are required to find gcd(a, b), with $m \geq 4$. Prove that $m < (\log_{1.5} a) - 1$. (*Hint*: Use the result of Exercise 38.)
40. a. Compute gcd(89, 55) and count the number of divisions required.
    b. Compute the upper bound on the number of divisions required for gcd(89, 55) using Equation (1).
    c. Compute the upper bound on the number of divisions required for gcd(89, 55) using the result of Exercise 39.
    d. The eighteenth-century French mathematician Gabriel Lamé proved that an upper bound on the number of division done by the Euclidean algorithm to find gcd(a, b) where $a > b$ is 5 times the number of decimal digits in b. Compute the upper bound on the number of divisions required for gcd(89, 55) using Lamé's theorem.

CHAPTER 3 REVIEW

TERMINOLOGY

analysis of algorithms (p. 203)
Backus–Naur form (BNF) (p. 163)
binary search algorithm (p. 169)
binary string (p. 163)
characteristic equation of a recurrence relation (p.190)
closed–form solution (p. 180)
concatenation (p. 163)
constant coefficient recurrence relation (p. 182)
divide-and-conquer algorithm (p. 208)
divide-and-conquer recurrence relation (p. 193)
empty string (p. 163)
Fibonacci sequence (p. 159)
first-order recurrence relation (p. 182)
homogeneous recurrence relation (p. 182)
index of summation (p. 182)
inductive definition (p. 158)
linear recurrence relation (p. 182)
palindrome (p. 163)
recurrence relation (p. 159)
recursive definition (p. 158)
second-order recurrence relation (p. 188)
selection sort algorithm (p. 168)
sequence (infinite sequence) (p. 158)
sequential search algorithm (p. 204)
solving a recurrence relation (p. 180)
structural induction (p. 164)
summation notation (p. 182)
upper bound (p. 210)