CHAPTER 1 COMPUTER EXERCISES:

1. How do you get your computing platform to test whether a (logical or mathematical) statement is true or false? Perform such a test with the following statements and compare with the actual truth values:
   (a) $3 < 6$
   (b) $-5 \geq 0$
   (c) $x^2 + y^2 = 25$, given that $x = 3$ and $y = 4$.

2. Use your computing platform to compute the truth values of each of the following compound statements, and compare with the actual truth values:
   (a) $(3 < 6) \wedge (-5 \geq 0)$
   (b) $(3 < 6) \rightarrow (-5 \leq 0)$
   (c) $\sim (x\sqrt{y} + 5 \geq 2) \vee [(x^3 + y^3 > z^3) \rightarrow (|x - y| > |z - y|)]$ given that $x = 3$, $y = 4$, and $z = 5$.
   (d) $[(3 < 6) \oplus (-5 \leq 0)] \leftrightarrow [((-3)^2 = 9) \rightarrow (8 \div 4 > 4)]$

NOTE: Implications are often used in computer programming with a syntax such as:[1]

```
IF <CONDITION A>  THEN  <DO TASK A>
```

This is not actually a statement, but rather a conditional command. In the course of a program, this command will check the truth value of <CONDITION A>, and if it is true, then <TASK A> will be executed, otherwise, nothing is done. Most computing platforms support more elaborate versions of the above basic syntax. For example a syntax such as:

```
IF <CONDITION A>  THEN  <DO TASK A>,
ELSE IF <CONDITION B>   <DO TASK B>,
ELSE IF <CONDITION C>   <DO TASK C>,
                  …
       ELSE <DO DEFAULT TASK>
                  …
```

would work as follows: first check if <CONDITION A> is satisfied, if it is, perform <TASK A>, and bypass the remaining "ELSE" lines. If <CONDITION A> fails, move on to check the next condition in the list (`<CONDITION B>`), if it is satisfied, perform the corresponding task (`<TASK B>`), and bypass the remaining "ELSE" lines. Continue, in this way, moving down the "ELSE IF"'s as necessary. The final "ELSE" line is optional; it allows a default task to be performed in case all of the previously listed conditions had failed.

The tasks can be either single commands, a sequence of commands, or a separate program (a subroutine). For example, suppose that variables `x` = 6, `y` = –2, and `z` = 8 are stored in the current workspace. The following command:[2]

```
IF   z/y > x
      THEN set z = 2*z
      ELSE set x = x - 3 and y = -y
```

would check that the condition `z/y > x (8/-2 > 6)` is false, and so the "ELSE" task would reset `x` to be its previous value less 3, or to `x = 3`, and `y` to the negative of its previously stored value, or `y` = 2. If this same command were now encountered again, the condition `z/y > x (8/2 > 3)` would now test true, so the "THEN" task would cause `z` to be reset to twice its currently stored value (of 8), so `z = 16.`

3. (a) Check by hand what the values of the variables `x`, `y`, and `z` will be after the following set of tasks are repeatedly run six times with initial variable values: `x` = 1, `y` = 1, and `z` = 2:

```
IF   (z ≥ 2*x + 4) OR  (y < x + 10)
     THEN set x = y, y = z, z = x + y
```

   (b) Confirm these hand-computed results by running the same code on your computing platform three times.
   (c) The IF condition in the above code is a disjunction of two conditions. Explain why if this program is repeated a sufficiently large number of times, this disjunction becomes a tautology.

4. Repeat Parts (a) and (b) of Exercise 3 if the code is modified as follows:

```
IF   (z < x + y) OR  (z/2 ≤ x)
     THEN set x = y, y = z, z = x + y
```

[1] Different computing platforms will have varying syntaxes and constructions, but the concepts behind the main programming constructions are very similar. We will adopt and adhere to an intuitive *pseudocode*, which will be summarized in Appendix A at the end of the book.
[2] We follow the computer programming custom of allowing a single equals sign to represent the operation of assignment. Thus, for example, if `z` is stored as 2, the mathematical equation `z = 2*z` makes no sense; but we take it to mean reassign `z` to be 2 times its old value (i.e., `z` will now be stored as 4).

```
ELSE set z = z – 1, x = x + 1
```

NOTE: One of the important strengths of computing platforms is their ability to perform a task repeatedly, and to update variables in the process. The action of performing such a repeated set of commands is called a **loop**, and we will introduce two sorts of loops. A **for loop** is a loop where the number of iterations is made completely explicit, usually by giving the range (and possibly also the steps) of the counter variable. For example, the following for loop (in our default pseudocode language) would add up the numbers 1, 2, 3, ⋯ ,10.

```
SUM = 0  (initialize the sum)
FOR K = 1 TO 10
    SUM = SUM + K  (update the sum)
END
```

The general syntax of a for loop is as follows:

```
FOR  N = <START> TO <LAST> INCREMENT = <GAP>
    <DO TASKS>
END
```

In the preceding example, we had <START> = 1, <LAST> = 10, and since the increment was (the default value) 1, it did not need to be specified. The <DO TASKS> portion of the loop, called the **body** of the loop, can be any set of commands that will be executed in each iteration of the loop. The END construct is required after the body to tell the platform when the loop is closed. The variable N in the above syntax is called the **counter** of the loop, and it may be included in the body of the loop (as it was in the previous example, where it was denoted by K). The way the loop works is that the counter is first set at its starting value <START>, then the commands in the body <DO TASKS> are executed in order, then the counter gets incremented by the INCREMENT <GAP>, and the commands in the body are again executed. This continues until the counter has reached its <LAST> value.

For example the following for loop could be used to compute the sum $4^2 + 6^2 + 8^2 + \cdots + 26^2$:

```
SUM = 0  (initialize the sum)
FOR K = 4 TO 26  INCREMENT = 2
    SUM = SUM + K^2  (update the sum)
END
DISPLAY(SUM)
```

(The last command after the loop is used to display the final value SUM, i.e., the desired sum.)

In Computer Exercises 5–6, for each part, write a for loop that will compute (as its only output) each of the following finite sums. Run it and give the answer (= the output).

5. (a) $1 + 2 + 3 + 4 + \cdots + 9999$. (b) $500 + 501 + 502 + \cdots + 1500$.
   (c) $500^2 + 501^2 + 502^2 + \cdots + 1500^2$. (d) $500 + 501^2 + 502 + 503^2 + \cdots + 1500$.

   Suggestion: For Part (d), the exponent changes at each iteration. Notice that it toggles between 1 and 2. One way to encode the toggle would be to include an assignment line like: `exponent = 3 – exponent` in the body of the loop. A simpler way would be to use an if branch within the body of the loop.

6. (a) $1 + 2 + 3 + 4 + \cdots + 20000$. (b) $50 + 51 + 52 + \cdots + 5050$.
   (c) $50^3 + 51^3 + 52^3 + \cdots + 2000^3$. (d) $50 + 51^3 + 52 + 53^3 + \cdots + 2000$.

7. (*Compound Interest versus Simple Interest*) (a) Suppose that your family has just discovered a savings account that was started by an ancient ancestor 240 years ago. The account was started with \$10 and paid 6.5% interest compounded annually. What is the account balance today?
   (b) Suppose instead that your ancestor had invested the \$10 in an account that paid 8% simple interest. This means that each year, the interest earned was 8% of the principal (i.e., 8% of \$10, or 80¢). Without using any loop (just basic arithmetic) what would the account balance of this account be today—240 years later?

8. (*Retirement Annuities*) (a) Suppose that Jackie sets up a retirement annuity that pays r = 7.5% interest, compounded annually. Starting when she turns 30 years old, Jackie deposits \$3500 each year, and plans to continue this until her 65th birthday (when she makes her last deposit and plans to retire). How much will Jackie's annuity be worth then?
   (b) Repeat Part (a) with r changed to 9%.
   (c) Repeat Part (a) with r changed to 12%.

9. (*Retirement Annuities*) (a) Suppose that Randall sets up a 401(k) retirement annuity that pays r = 6% interest, compounded annually. When he turns 35 years old, Randall deposits \$3500, and each subsequent year he plans to deposit 5% more than the previous year (due to planned raises and better money management). He continues this until his 65th birthday (when he makes his last deposit and plans to retire). How much will Randall's annuity be worth then?

(b) Repeat Part (a) if the annual deposits are doubled.
(c) Repeat Part (a) if the annual interest rate is doubled.
(d) Repeat Part (a) if he starts the annuity 10 years earlier (when he turns 25), but still continues until his 65th birthday.

In many situations, we do not know in advance how many iterations are needed, but rather we would like the iterations to stop when a certain condition is met. In contrast to a for loop, a while loop is suitable for such a purpose. The general syntax of a **while loop** is shown below:

```
WHILE <CONDITION>
    <DO TASKS>
END
```

Here, `<DO TASKS>` can be any set of commands as in a for loop. When encountered in a program code, the above while loop will function as follows: First the truth value of `<CONDITION>` is checked. If it tests false, the while loop is bypassed with no action having been taken. If it tests true, then all the commands in the body `<DO TASKS>` are performed, and `<CONDITION>` is reevaluated. If it tests false now, the while loop is exited, while if `<CONDITION>` tests true, then `<DO TASKS>` is performed once again, and the program will go back again to check the truth value of `<CONDITION>`. This process continues.[3]

For example the following for loop could be used to compute the sum $4^2 + 6^2 + 8^2 + \cdots$ by continuing to add terms until the sum first equals or exceeds 1000.

```
SUM = 0  (initialize the sum)
TERM = 4 (initialize the term)
WHILE SUM < 1000
    SUM = SUM + K^2  (update the sum)
    TERM = TERM + 2  (increment term)
END
DISPLAY(SUM)
```

Notice one important difference with for loops: In while loops the counter term needed to be initialized and incremented inside the loop. This was done automatically with for loops.

In Exercises 10–11, for each part, write a while loop that will add up the terms of the given sequence until the sum first exceeds or equals the number M that is given. Your loop should have (exactly) two outputs: both the number of terms that were added up, and the sum of these terms. Run it and give the answers (= the output).

10. (a) $1 + 2 + 3 + 4 + \cdots;\ M = 10{,}000.$ (b) $500 + 501 + 502 + \cdots;\ M = 1{,}000{,}000.$
(c) $500^2 + 501^2 + 502^2 + \cdots;\ M = 1{,}000{,}000.$ (d) $500 + 501^2 + 502 + 503^2 \cdots;\ M = 1{,}000{,}000.$

11. (a) $1 + 2 + 3 + 4 + \cdots;\ M = 2{,}000{,}000.$ (b) $50 + 51 + 52 + \cdots;\ M = 2{,}000{,}000.$
(c) $50^3 + 51^3 + 52^3 + \cdots;\ M = 2{,}000{,}000{,}000.$ (d) $50 + 51^3 + 52 + 53^3 + \cdots;\ M = 2{,}000{,}000{,}000.$

12. (*Compound Interest*) (a) Suppose that $1000 is invested in a long-term CD that pays 7% interest compounded annually. Write a while loop to determine the number of years that it would take the money to triple (to equal or exceed $3000), and the resulting balance after this number of years.
(b) Re-do Part (a) if the amount invested is changed to $1 million.
(c) Re-do Part (a) with the annual interest rate changed to 9%.
(d) Re-do Part (a) with the annual interest rate changed to 5%.

13. (*Retirement Annuities*) For each part below, re-do Example 3 with the indicated changes (the changes indicated for each part apply only to that part).
(a) Suppose that Michael instead decides to retire when his annuity reaches or exceeds $500,000.
(b) Suppose that the interest rate is 10% (rather than 9%).
(c) Suppose that rather than making the same deposit every year, because of regular raises and better money management, Michael's annual deposits grow by 6% each year. So the first year he puts in $5000, the second year $5300, the third year $5618, etc.

14. (*Retirement Annuities*) Suppose that Jocelyn sets up a 401(k) that pays r = 10% interest, compounded annually.

[3] **Caution:** Unlike with a for loop, there is always a danger that a while loop can enter into an *infinite loop*. You should learn how to exit from an infinite loop in case your programs accidentally enter into one. One common strategy that works on several platforms is to enter the CTRL-C command.

Starting when she turns 36 years old, Jocelyn deposits $10,000 each year, and plans to continue this until her annuity meets or exceeds $1 million. How many years will it take for this to happen?
(a) Suppose instead that Jocelyn decides to retire when her annuity reaches or exceeds $1.5 million.
(b) Suppose instead that the interest rate is 12% (rather than 10%).
(c) Suppose that rather than making the same deposit every year, because of regular raises and better money management, Jocelyn's annual deposits grow by $1000 each year. So the first year she puts in $10,000, the second year $11,000, the third year $12,000, etc.

15. (*Paying off a Loan*) (a) Suppose that Gomez takes out a loan for $22,000 to buy a new SUV (after trading in his car). The bank charges 6% annual interest on the unpaid balance, compounded monthly (that's 0.5% interest each month). If Gomez pays $300 at the end of each month, how many months will it take him to pay off his loan? What is the amount of his last payment? His last payment will just cover the unpaid balance so will likely be less than $300.
(b) Suppose that, in writing a while loop to solve this problem, you accidentally typed 220000 (rather than 22000); i.e., you added an extra zero to the loan amount. Explain what would happen. If you don't know what would happen, try it out.

16. (*Paying off a Loan*) (a) Suppose that the Jones family borrows $420,000 to buy a house. The bank charges 6% annual interest on the unpaid balance, compounded monthly. If the Jones's pay $3000 at the end of each month, how long will it take them to pay off the loan, and what will the amount of their last payment be? Their last payment will just cover the unpaid balance so will likely be less than $3000.
(b) Suppose that, in entering a while loop to solve this problem you accidentally typed 4200000 (rather than 420000); i.e., you added an extra zero to the loan amount. Explain what would happen. If you do not know what would happen, try it out.

CHAPTER 3 COMPUTER EXERCISES:

NOTE: Finite sets can be stored as objects on most computing platforms. Depending on your particular platform, how a set gets stored might depend on the nature of the elements. If the elements are numbers, the set is probably stored as a list or vector, where the order does not matter (nor should any duplicated entries). If the elements are text strings, such as in the set {Red, White, Blue}, then the storage will require a different type of data structure. In the computer exercises that follow, the objective will be to learn how to store both types of finite sets on your platform and to learn how to perform the set operations. We point out that the size of a set being stored on a given computer platform is limited by the available memory. Also, there are different grades of efficiency for storing given sets. At this juncture, we will be working with small sets so memory and efficiency issues will not be relevant. These issues are important for many practical problems, however, and we will revisit these topics as needs arise.

In Exercises 1–4 below, in addition to learning how to store the sets given, you should also find out how to perform (at least) the following set operations: union, intersection, and (relative) complement.

1. Store the sets of (regular) Exercise 1 of this section on your computing platform, and perform the set operations asked for in Parts (a) through (f) of that exercise.

2. Store the sets of (regular) Exercise 2 of this section on your computing platform, and perform the set operations asked for in Parts (a) through (f) of that exercise.

3. Store the sets of (regular) Exercise 3 of this section on your computing platform, and perform the set operations asked for in Parts (a) through (f) of that exercise.

4. Store the sets of (regular) Exercise 4 of this section on your computing platform, and perform the set operations asked for in Parts (a) through (f) of that exercise.

5. Find out how your computing platform can test whether a number is an element of a set of numbers.[4] Similarly, find out how you can test the subset $(A \subseteq B)$ and equality $(A = B)$ relationships between two sets. Next, using the sets A, B, C, and U of (regular) Exercise 4, run through and test each of the identities in Parts (a) through (d) of (regular) Exercise 11.

6. Read Exercise 5 and then again using the sets A, B, C, and U of (regular) Exercise 4, run through and test each of the identities in Parts (a) through (d) of (regular) Exercise 12.

[4] Try to avoid the obvious brute force approach of using a for loop to run through the elements of the set, one-by-one, checking whether the given element equals the set element. Look for a more elegant approach, or better yet, a built-in function that will do the job.

CHAPTER 7 COMPUTER EXERCISES:

NOTE: If your computing platform is a floating point arithmetic system, it may allow you up to only 15 or so significant digits of accuracy. Symbolic systems allow for much greater precision, being able to handle 100s or 1000s of significant digits. Some platforms allow the user to choose if they wish to work in floating point or symbolic arithmetic, but will work in floating point arithmetic by default since operations are faster and usually sufficiently accurate for general purposes. If you have access to only a floating point system, you should keep these limitations in mind when you do computer calculations with large integers. Some particular questions below may need to be skipped or modified so the numbers are of a manageable size. See Section 4.4 for more on the important differences between floating point and symbolic computing systems.

1. (*Empirical Comparison of the Iterative versus Recursive Programs for the Fibonacci Sequence*) Write programs (in your computing platform) `y = fibonacci(n)`, `y = fibonacciv2(n)`, corresponding to the programs of the same names/syntax that were presented in Example 6. Run each program with inputs n = 2, 4, 6, 8, … until it takes more than 2 minutes to execute. Record the runtimes and compare them with the complexity analysis that was presented in Example 6.

2. (*Empirical Comparison of the Iterative versus Recursive Programs*) Write programs (in your computing platform) corresponding to the iterative and recursive programs of Example 12. Run each program with inputs n = 2, 4, 6, 8, … until it takes more than 2 minutes to execute. Record the runtimes and comment on the relative complexity of the two algorithms.

3. (*Empirical Examination of the Cofactor Expansion Program*) Write a program (in your computing platform) `y = DetCofactor(A)`, corresponding to the program of the same name/syntax that was presented in Example 9. Run the program with on some $n \times n$ matrices whose entries are taken (preferably "randomly") in the range {0, 1, 2, …, 9}, with n = 2, 4, 6, 8, … until it takes more than 2 minutes to execute. Record the runtimes and compare them with the complexity analysis that was presented in Example 9.
   Note: Although the topic of random integer generation will be formally discussed in Chapter 6, most computing platforms have utilities (often under the name `rand`) that at each call will generate a "pseudorandom" real number in the range (0,1). If you can find such a utility, use the composition floor(`rand`) to generate a random integer in the range {0, 1, 2, …, 9}.

4. (*Empirical Comparison of the Iterative versus Recursive Programs for Finding the Maximum in a List of Numbers*) Write programs (in your computing platform) `Max = IterativeMax(vec)`, and `Max = RecursiveMax(vec)` each inputting a vector (i.e., an ordered list) `vec` of real numbers, and outputting the maximum number `Max` in the list. The first program is the iterative algorithm given in (ordinary) Exercise 27, and the latter is the recursive algorithm of Part (a) of that exercise. Run the program with on some length n vectors whose entries are taken (preferably "randomly") to be real numbers in the range (0,n), with n = 10, 100, 1000, etc., until either n reaches 1 million or the computation takes more than two minutes. Record the runtimes and compare with the answer to Part (b) of Exercise 27.
   Note: See the note to the previous exercise. If you can find such a `rand` utility, generate the entries of the length n vector as `n*rand`.

5. (*Recursive Algorithm for Towers of Hanoi Puzzle Solutions*) (a) Write a recursive program that will delineate the moves needed to solve the Towers of Hanoi puzzle (based on the solution of Example 2) with n disks. The syntax should be: `Moves = TowerOfHanoiSol(n)`, where the input `n` is the number of disks, and the output `Moves` is a 3 column matrix whose rows indicate the sequence of moves (in order) that will solve the Towers of Hanoi puzzle with n disks. Each row of `Moves` will be a vector of three integers: (d, pStart, pEnd), where the first entry d indicates the disk that will be moved, the second entry pStart the peg from where the disk will be moved, and the third entry pEnd the peg to where the disk will be moved. For definiteness assume that the smallest disk has index d = 1, the next smallest has index d = 2, and so forth.
   (b) Run your program of Part (a) with n = 1, 2, 3, and 4, and verify the correctness of the ouputs.

6. (*Recursive Algorithm for a Geometric Tiling*) Recall that in Example 1 of Chapter 6, it was proved by induction that any $2^n \times 2^n$ grid of squares with a single square removed can be tiled by 3-square L-shaped tiles. The proof also provided a recursive scheme to achieve such a tiling.
   (a) (*Non-Graphical Interface Implementation*) Identify the grid as a $2^n \times 2^n$ matrix G whose rows and columns are indexed from 1 to 2^n. Identify each 3-square L-shaped tile as a triple, (i, j, α), where the first two indices give the row and column index of the central square (so $1 \le i, j \le 2^n$), and $\alpha \in \{1,2,3,4\}$ indicates the orientation of the tile (once the central square of the tile has its location specified, there are four possible ways to lay the tile in a grid). Note that when the central square is on a border, some values of α are infeasible. Write a recursive program `Tiles = ThreeSquareLTiling(n, DelSq)`, whose first input is a positive integer n, whose second input `DelSq` is the vector of indices (i,j) of a deleted square in the $2^n \times 2^n$ matrix (representing a grid) G, and whose output `Tiles` is a three-column matrix whose rows represent a tiling of G with the prescribed single deleted square.
   (b) Run your program of Part (a) with n = 1, 2, and 3, each with two different deleted squares and graphically verify the correctness of the ouputs.
   (c) (*Graphical Interface Implementation–For Readers Who Have Experience with Computer Graphics*) Create a

graphically enhanced version of the program of Part (a). The inputs and outputs should be the same, but the program should also produce a graphic of the grid with the tiles indicated in some easy to read fashion.
(d) Run your program of Part (c) with n = 1, 2, and 3, each with two different deleted squares and graphically verify the correctness of the ouputs.
Suggestion: For Part (a), as the algorithm progresses, let the matrix *G* representing the grid have a 0 entry if the corresponding square is not yet covered by a tile, and a 1 entry if it is covered or is the deleted square. One simple way to implement Part (c) would be to indicate each tile by a skeletal L drawn within the tile in heavy line font, and to have the grid lines in much lighter dotted line font.

CHAPTER 8 COMPUTER EXERCISES:

NOTE: If your computing platform is a floating point arithmetic system, it may only allow you up to 15 or so significant digits of accuracy. Symbolic systems allow for much greater precision, being able to handle 100s or 1000s of significant digits. Some platforms allow the user to choose if they wish to work in floating point or symbolic arithmetic, but will work in floating point arithmetic by default since operations are faster and usually sufficiently accurate for general purposes. If you only have access to a floating point system, you should keep these limitations in mind when you do computer calculations with large integers. Some particular questions below may need to be skipped or modified so the numbers are of a manageable size. See Section 4.4 for more on the important differences between floating point and symbolic computing systems.

1. (*Program for Check if a Positive Integer is Prime*) (a) Write a program `y = PrimeCheck(n)` that inputs an integer `n` > 1, and outputs an integer `y` that is 1 if `n` is a prime number and 0 if it is not. The method used should be a brute force check to see whether n has any positive integer factor *k*, checking all values of *k* (if necessary) up to floor($\sqrt{n}$).
(b) Run your program with each of the following input values: n = 30, 31, 487, 8893, 987654323, 131317171919.
(c) Assuming that your computing platform can perform 1 billion divisions per second (and assuming that the rest of the program in Part (a) takes negligible time), what is the largest number of digits an inputted integer n could have so that the program could be guaranteed to execute in less than 1 minute?
(d) Under the assumption of Part (c), how long could it take for the program in Part (a) to check whether a 100 digit integer is prime?
Note: Of course, the program could execute very fast if a small prime factor is found quickly. A prime input would always take the most time since the full range of *k* values would need to be checked. There are some efficiency enhancements that we could incorporate into the above program: For example, after it is checked that 2 is not a factor, we need only check odd integers after 2. Such a fix could cut the runtimes essentially in half, but there are more efficient (and sophisticated) prime checking algorithms than such brute-force methods. For more on this interesting area, we refer the reader to [BaSh-96].

2. (*Program for Prime Factorization of Positive Integers*) (a) Write a program `FactorList = PrimeFactors(n)` that inputs an integer `n` > 1, and outputs a vector `FactorList` that lists all of the prime factors, from smallest to largest, and with repetitions for multiple factors. For example, since the prime factorization of 24 is $2^3 \cdot 3$, the output of `PrimeFactors(24)` should be the vector [2 2 2 3]. Starting with `k` = 2, and running `k` through successive integers, the method should check to see whether `n` has k as a factor. If it does, then the k should be appended to the `FactorList` output vector. We then replace `n` with `n/k`, and continue to check whether k divides into (the new) n until it no longer does, and then we move on to update k to k + 1. With this scheme, only prime factors will be found. (Why?) Also, we may stop as soon as k reaches the current value of floor($\sqrt{n}$).
(b) Run your program with each of the following input values: n = 30, 31, 487, 8893 987654323, 131317171919.
Note: Although there are faster algorithms, the factorization problem of this computer exercise is more untractable than the primality checking algorithm of the preceding exercise. In the language of Chapter 7, a polynomial time algorithm has been found for the primality checking problem, but it is strongly believed that there can exist no such algorithm for the prime factorization problem. This latter fact is the basis for the success of the widely used RSA cryptosystem that we will study in Section 4.5. See also the note of the preceding exercise, and the reference given there.

3. (*Program for Finding the Next Prime*) (a) Write a program `p = NextPrime(n)` that inputs an integer `n` > 1, and outputs the smallest prime number `p` ≥ n.
(b) Run your program with each of the following input values: n = 8, 30, 32, 487, 8899 987654321, 131317171919.
Suggestion: Call on the program of Computer Exercise 1.

4. (*Program for the Division Algorithm*) (a) Write a program `(q,r) = DivAlg(a,d)` that inputs an integer `a` = the dividend, and a positive integer `d` = the divisor. The output will be the (unique) integers `q` = the quotient, and `r` = the remainder, satisfying `a` = `dq` + `r`, with 0 ≤ `r` < `a` (as guaranteed by Proposition 6).
(b) Run your program with each of the following pairs of input values: `(a,d)` = (5, 2), (501, 13), (1848, 18), (123456, 321).
Suggestion: Use the idea of Example 4.

NOTE: (**`mod` *Function on Computing Platforms***) Most computing platforms have some sort of remainder or modular integer converter; this will be particularly useful for modular arithmetic computations. The next exercise will ask you to either find such a function on your platform or (if you cannot find one or there is none) to write a (simple) program for one.

5. (a) Either find a program on your computing platform that performs as follows, or write one of you own: Write a program with syntax: `b = mod(a,m)` that will take as inputs an integer `a`, and an integer `m` > 0 (the modulus). The output should be a nonnegative integer `b`, with $0 \le b < m$, that is congruent to `a` (mod `m`). In other words, `b` should be the remainder when `a` is divided by `m` using the division algorithm.
(b) Use this function to redo the hand calculations that were asked in (ordinary) Exercises 21 and 22.
Suggestion: (For Part (b)) In case your system works in floating point arithmetic (or if you are not sure), in Parts (e) of (ordinary) Exercises 21 and 21, directly evaluating the very large ordinary integers as inputs in the mod function would result in inaccurate results. Instead, for example, to evaluate $21^{223} \pmod{24}$, start by iteratively computing `b1 = mod(21^2,m)`, `b2 = mod(b1^2,m)` $\equiv 21^4 = 21^{2^2}$, `b3 = mod(b2^2,m)` $\equiv 21^8 = 21^{2^3}$, `b4 = mod(b3^2,m)` $\equiv 21^{16} = 21^{2^4}$, etc., until we get to the largest such exponent less than (or equal to) the desired exponent. Then multiply out some of the appropriate intermediate results (using the `mod` function) to obtain the desired power. This idea can be streamlined into a very fast and effective modular exponentiation algorithm, and this will be done in Section 4.2.

6. (*Program for the Euclidean Algorithm*) (a) Write a program `d = EuclidAlg(a,b)` that inputs two positive integers `a` and `b`, and outputs `d` = gcd(`a`,`b`), computed using the Euclidean algorithm (Algorithm 1).
(b) Check your program with the results of Example 5, and then run it on each of the following pairs of input values: `(a,b)` = (525, 223), (12364, 9867), (1234567890, 0987654321), (13131717191919, 191917171313).

7. (*Program for the Euler Phi Function*) (a) Write a program `y = EulerPhi(n)` that inputs an integer n > 1, and outputs `y` = $\phi(n)$.
(b) Check your program with the results of (ordinary) Exercise 23, and then use it to compute $\phi(n)$ for each of the following values of n: 18,365, 222,651, 1,847,773, 22,991,877.
Suggestion: Call on the program of Computer Exercise 2 for factoring integers and use the formula for $\phi(n)$ given in Proposition 3.20. Since this program relies on factorization, it is not intended to be used with large integers (of say 10 digits or more).

8. (*Program for Computing Orders*) (a) Write a program `k = Order(a,n)` that inputs two relatively prime positive integers `a`, and `n`, with `a` < `n`, and outputs `k` = $\text{ord}_n(a)$.
(b) Check your program with the results of (ordinary) Exercise 33, and then use it to compute these two orders: $\text{ord}_{1807}(3)$, $\text{ord}_{10543}(54)$.
Suggestion: This can either be done with a brute-force approach (of computing successive powers of `a` until we reach 1), or using Proposition 3.22 (and calling on the program of Computer Exercise 1). In either case, the programs are not suitable for large integers.

9. (*Program for Finding Primitive Roots*) (a) Write a program `pRoot = SmallestPrimitiveRoot(n)` that inputs a positive integer `n`, that is of the form listed in Theorem 15, and that outputs the smallest primitive root of `n` (guaranteed to exist by Theorem 15). It is fine to call on the program `Order` of the preceding exercise.
(b) Check your program with the results of (ordinary) Exercise 37(a)–(c), and then run it on the following values of n: 8893, 17786, 123457, 246914.
(c) Write a related program `PRoots = AllPrimitiveRoots(n)` that inputs a positive integer `n`, that is of the form listed in Theorem 15, and that outputs a vector `PRoots` of all of the $\phi(\phi(n))$ primitive roots of `n` (see Theorem 15).
(d) Run your program of Part (c) on each of the inputs of Part (b).
Suggestion: For Part (c), make use of (ordinary) Exercise 49.

CHAPTER 9 COMPUTER EXERCISES:

NOTE: There are two natural *data structures* for storing base b expansions: either as vectors or as strings. Strings are only appropriate in case the base b "digits" are single characters: This includes the cases $b \le 10$ and $b = 16$ (due to our special hexadecimal notation). Vectors tend to be more amenable to writing programs, but strings display more efficiently. The reader should contemplate both possibilities on his/her particular computing platform and decide which of these options (or perhaps another option) would be most suitable for the exercises and applications of this section.

1. Write a program `n = bin2int(v)` that will take as input a vector `v` for a binary expansion (zeros and/or ones), and will output its corresponding equivalent decimal integer `n`. Perform, by hand, the corresponding conversions for the binary strings [1011], [11111], and [1011110], and run your program on these inputs (debug, as necessary).

2. Write a program `n = oct2int(v)` that will take as input a vector `v` for an octal (base 8) expansion, and will output its corresponding equivalent decimal integer `n`. Perform, by hand, the corresponding conversions for the octal expansions [5027], [23456], and [7031410], and run your program on these inputs (debug, as necessary).

3. Write a program `n = hex2int(v)` that will take as input a vector `v` for a hexadecimal (base 16) expansion, and will output its corresponding equivalent decimal integer `n`. Perform, by hand, the corresponding conversions for the hexadecimal expansions [8B7], [1AEE], and [AAAA6], and run your program on these inputs (debug, as necessary).
Note: The reader may wish to intead use vectors of integers in the range 0–15 for hexadecimal sequences.

4. Write a program `n = base92int(v)` that will take as input a vector `v` viewed as a base 9 expansion, and will output its corresponding equivalent decimal integer `n`. Perform, by hand, the corresponding conversions for the base 9 expansions [847], [1444], and [65626], and run your program on these inputs (debug, as necessary).

5. Write a program `n = baseb2int(v,b)` that will take two inputs: a vector `v` for a base `b` expansion, and, `b` an integer greater than 1 (for the base of the expansion). The output will be the corresponding equivalent decimal integer `n`. Run your program on each of the expansions of ordinary Exercise 5 (debug, as necessary).

6. Write a program `v = int2bin(n)` that will take as input a nonnegative integer `n`, and will output its binary expansion vector `v` using Algorithm 1. Perform, by hand, the corresponding conversions to binary expansions for n = 8, 107, 327, and 12,557, and run your program on these inputs (debug, as necessary).

7. Write a program `v = int2oct(n)` that will take as input a nonnegative integer `n`, and will output its octal (base 8) expansion vector `v` using Algorithm 1. Perform, by hand, the corresponding conversions to binary expansions for n = 8, 107, 327, and 12,557, and run your program on these inputs (debug, as necessary).

8. Write a program `v = int2hex(n)` that will take as input a nonnegative integer `n`, and will output its hexadecimal (base 16) expansion vector `v` using Algorithm 1. Perform, by hand, the corresponding conversions to binary expansions for n = 8, 107, 327, and 12,557, and run your program on these inputs (debug, as necessary).
Note: Read Computer Exercise 3.

9. Write a program `v = int2baseb(n, b)` that will take as inputs a nonnegative integer `n` and an integer `b` (the base) greater than one. The output will be the base `b` expansion vector `v` of the integer `n`, determined by using Algorithm 1. Perform, by hand, the corresponding conversions for n = 8, 107, 327, and 12,557, with each of the bases 2, 8, 16, and run your program on these (14) inputs (debug, as necessary).

10. Write a program `w = bin_add(u,v)` that will take as inputs two vectors `u` and `v` representing binary (base 2) expansions, and will output the vector `w` representing the binary expansion of the sum `u + v`, computed using Algorithm 2. Perform, by hand, the binary additions [101] + [111], [110110] + [1010111], and run your program for these additions (debug, as necessary).

11. Write a program `w = hex_add(u,v)` that will take as inputs two vectors `u` and `v` representing hexadecimal (base 16) expansions (using digits from 0 to 15), and will output the vector `w` representing the hexadecimal expansion of the sum `u + v`, computed using Algorithm 2. Perform, by hand, the hexadecimal additions [C42] + [A1A], [86B4D] + [76A0C], and run your program for these additions (debug, as necessary).
Note: The reader may wish to instead use vectors of integers in the range 0–15 for hexadecimal sequences.

12. Write a program `w = baseb_add(u,v,b)` that will take as inputs two vectors `u` and `v` representing base `b` expansions, and a third input `b` (the base) being an integer greater than 1. The output will be the vector `w` representing the base `b` expansion of the sum `u + v`, computed using Algorithm 2. Run your program on the base *b* subtractions of ordinary Exercise 15 (debug, as necessary).

13. Write a program `w = bin_sub(u,v)` that will take as inputs two vectors `u` and `v` representing binary (base 2) expansions where `u ≥ v`, and will output the vector `w` representing the binary expansion of the difference `u – v`, computed using Algorithm 3. Perform, by hand, the binary subtractions [101] – [011], [110110] – [101011], and run your program on them (debug, as necessary).

14. Write a program `w = hex_sub(u,v)` that will take as inputs two vectors `u` and `v` representing hexadecimal (base 16) expansions where `u ≥ v`, and will output the vector `w` representing the hexadecimal expansion of the difference `u – v`, computed using Algorithm 3. Perform, by hand, the hexadecimal subtractions [A42] – [A1A], [86A4D] + [76C8B], and run your program on them (debug, as necessary).
Note: The reader may wish to use instead vectors of integers in the range 0–15 for hexadecimal sequences.

15. Write a program `w = baseb_sub(u,v,b)` that will take as inputs two vectors `u` and `v` representing base `b` expansions where `u ≥ v`, and a third input `b` (the base) being an integer greater than 1. The output will be the vector `w` representing the base `b` expansion of the difference `u – v`, computed using Algorithm 3. Run your program on the base *b* subtractions of ordinary Exercise 17 (debug, as necessary).

16. Write a program `w = bin_mult(u,v)` that will take as inputs two vectors `u` and `v` representing binary (base 2) expansions, and will output the vector `w` representing the binary expansion of the product `u × v`, computed using Algorithm 4. Perform, by hand, the binary multiplications [101] × [111], [110110] × [1010111], and run your program for these multiplications (debug, as necessary).

17. Write a program `w = hex_mult(u,v)` that will take as inputs two vectors `u` and `v` representing hexadecimal (base 16) expansions (using digits from 0 to 15), and will output the vector `w` representing the hexadecimal expansion of the product `u × v`, computed using Algorithm 4. Perform, by hand, the hexadecimal multiplications [C42] × [A1A], [86B4D] × [76A0C], and run your program for these multiplications (debug, as necessary).
Note: The reader may wish to instead use vectors of integers in the range 0–15 for hexadecimal sequences.

18. Write a program `w = baseb_mult(u,v,b)` that will take as inputs two vectors `u` and `v` representing base `b` expansions, and a third input `b` (the base) being an integer greater than 1. The output will be the vector `w` representing the base `b` expansion of the product `u × v`, computed using Algorithm 4. Run your program on the base *b* multiplications of ordinary Exercises 19, 21 (debug, as necessary).

CHAPTER 10 COMPUTER EXERCISES:

NOTE: We point out here that certain computing platforms are accurate to only 16 or so significant digits (in the default configuration) since they operate in so-called *floating point arithmetic*. Other platforms can handle arbitrarily large amounts of precision, but usually at a greater cost of computation time. For more about floating point arithmetic, see Chapter 5 of [Sta-05]. Unless you are aware of how to manipulate more than 16 digits of accuracy on your system, you should restrict your computer applications to this limit.

Most computing platforms have some sort of remainder or modular integer converter; this will be particularly useful for modular arithmetic computations. Recall that Computer Exercise 5 of Chapter 8 asked you to either find such a function on your platform or (if you cannot find one or there is none) to write a (simple) program for one. The reader should do this before beginning to work on the following set.

1. (*Brute-Force Program for Finding Modular Inverses*) (a) Write a program with syntax: `ainv = modinv_bf(a, m)` that will take as inputs a modular integer `a`, and an integer `m > 1` (the corresponding modulus). If `a` has an inverse (mod `m`), the output `ainv` will be the corresponding (unique) inverse, if `a` has no inverse, then there will be no output for `ainv`, but only a message to the effect that "there is no inverse mod `m`." The programming should be done by brute-force, i.e., checking through all of the integers `b` mod `m`, and multiplying these by `a` to see if we get 1 (mod `m`). If this ever happens, `b` will be the inverse, so we can output `ainv` as `b`, if not, then `a` has no inverse. You may save time in this search by using the necessary condition gcd(`a`,`m`) = 1, and gcd(`b`, `m`) = 1 for elements to be invertible (or be inverses).
(b) Use this program to re-do Exercises 15 through 18.
(c) Use this program to determine the following inverses, if they exist: $1335^{-1} \pmod{39467}$ and $87451^{-1} \pmod{139467}$.

2. (*Program for the Extended Euclidean Algorithm*) (a) Write a program with syntax: `outVec = ExtEucAlg(a,m)` that takes two inputs: `a` and `b`, which are positive integers such that `a ≥ b`. The output will consist of a length three vector `outVec` that has three integer components, `d`, `x`, and `y`, where `d` = gcd(`a`,`b`), and `x` and `y` satisfy $d = ax + by$ (as in the Algorithm 4). The program should follow Algorithm 4.
(b) Use your program to check the results of Example 14 (a).
(c) For each pair of integers *a* and *b* that we list here, use your program to compute gcd(*a*,*b*), and two integers *x*, and *y* such that $d = ax + by$. (i) $a = 8359$, $b = 4962$, (ii) $a = 95{,}243$, $b = 24{,}138$.
(c) Use your program to solve the equation $88243x + 16947y = 1$, for integers *x* and *y* (or to determine that such a solution does not exist).

3. (*Program for Finding Inverses with the Extended Euclidean Algorithm*) (a) Write a program with syntax: `ainv = modinv(a,b)` that has the same syntax, inputs and outputs as the program of Computer Exercise 1, but now the programming should use the extended Euclidean algorithm (Algorithm 4). Alternatively, your program can directly call on the one of Computer Exercise 2, if you have done that one.
(b) Use this program to re-do Exercises 9 through 12.
(c) Use this program to determine the following inverses, if they exist: $1335^{-1} \pmod{39467}$, and $87451^{-1} \pmod{139467}$.
(d) Compare the performance times of `modinv` and `modinv_bf` (of Computer Exercise 1) for the following pairs of inputs: a = 967, and (i) $m = 10{,}001$, (ii) $m = 100{,}001$, (iii) $m = 1{,}000{,}001$, (iv) $m = 10{,}000{,}001$.

4. (*Program for Solving General Congruences of the Form $ax \equiv c \pmod{m}$*) (a) Write a program with syntax: `SolVec = LinCongSolver(a,c,m)`, that will take as inputs the parameters `a`, `c`, and `m` that determine a single linear congruence $ax \equiv c \pmod{m}$, and will output a vector `SolVec` (possibly empty, if there are no solutions) containing all of the solutions of this congruence (mod `m`). In case gcd(`a`,`m`) > 1, the program should follow Algorithm 3. The program can be made simpler if it calls on the program `modinv` of Computer Exercise 3.
(b) Use your program to check the results of Example 4.11 and of Exercise for the Reader 4.9.
(c) Use this program to re-do Exercises 13 and 14.
(d) Use this program to re-do Exercises 19 and 20.

5. (*Program to Check ISBN-13 Numbers*) (a) Write a program with syntax: `check = ISBN13(vec)`, where the input is a vector `vec` of the 13 digits representing an ISBN-13 number (read Exercise 29 for the necessary background). For example, the ISBN-13 number 978-0-471-69738-1 would be inputted as the vector [9 7 8 0 4 7 1 6 9 7 3 8 1] (dashes are left out). The output `check` will be a string of text: either "Valid ISBN-13 number" if the vector entered does indeed correspond to a valid ISBN number (i.e., satisfies (12)) or "Invalid ISBN-13 number."
(b) Use this program in conjunction with a for loop to re-do Part (a) of Exercise 29.

6. (*Program to Check VISA Credit Card Numbers*) (a) Write a program with syntax: `check = VISA16(vec)`, where the input is a vector `vec` of the 16 digits representing a 16 digit VISA card number (read Exercise 31 for the necessary background). For example, the VISA card number 4784 5580 0246 1888 would be inputted as the vector [4 7 8 4 5 5 8 0 0 2 4 6 1 8 8 8]. The output `check` will be a string of text: either "Valid VISA card number" if vector entered does indeed correspond to a valid ISBN number (i.e., satisfies the congruence (14)), or "Invalid VISA card number."
(b) Use this program in conjunction with a for loop to re-do Part (a) of Exercise 31.

7. (*Program for Fast Modular Exponentiation*) (a) Write a program with syntax: `a = ModularExponentiation(b,x,m)` that will take as inputs a modular integer $b \not\equiv 0 \pmod{m}$, a positive integer exponent `x` and a corresponding modulus `m` > 1. The output `a` will be the unique nonnegative integer < `m` that satisfies $a \equiv b^x \pmod{m}$. The mechanics of the program should follow Algorithm 2.
(b) Use this program to re-do Exercise 11.
(c) Use this program to compute the following:
(i) $2^{1234567890} \pmod{169}$ and (ii) $12^{123456789012} \pmod{1865}$.

8. (*Brute-force Program for Finding Square Roots Modulo a Prime p*) Before doing this problem, you need to read Exercise 38. (a) Write a program with syntax: `sqrtvec = modsqrt_bf(a, p)` that will take as inputs a modular integer $a \not\equiv 0 \pmod{p}$, and an odd prime integer `p` > 2 (the corresponding modulus). If `a` has a square root (mod `p`), the output `sqrtvec` will be the length two vector containing the two square roots of `a`, if `a` has no square roots, then there will be no output for `sqrtvec`, but only a message to the effect that "there are no square roots mod `p`." The programming should be done by brute-force as suggested in Exercise 38, i.e., checking through all of the integers $\{0^2, 1^2, 2^2, \cdots, [(p-1)/2]^2\}$, to see if one of them is $\equiv a \pmod{p}$. If this happens, say $b^2 \equiv a \pmod{p}$, then the two square roots of a will be b and $-b \pmod{p}$, and they should be outputted into the vector `sqrtvec`.
(b) Use this program to re-do Parts (d) and (e) of Exercise 38.
(c) Use this program to re-do Parts (c) and (d) of Exercise 39.
(d) Use this program to re-do Part (b) of Exercise 40.

9. (*An Efficient Program for Finding Square Roots Modulo a Prime $p \equiv 3 \pmod{4}$*) Before doing this problem, you need to read Exercise 40. (a) Write a program with syntax: `sqrtvec = modsqrt_p3m4(a, p)` that will take as inputs a modular integer $a \not\equiv 0 \pmod{p}$, and prime integer $p \equiv 3 \pmod{4}$ (the corresponding modulus). If `a` has a square root (mod `p`), the output `sqrtvec` will be the length two vector containing the two square roots of `a`, if `a` has no square roots, then there will be no output for `sqrtvec`, but only a message to the effect that "there are no square roots mod `p`." The programming should be done using Proposition 6 (within Exercise 37).
(b) Use this program to re-do Part (b) of Exercise 40.
(c) For each of the following odd primes p, which are congruent to 3 (mod 4), compare the performance of your program `modsqrt_p3m4` with that for the brute-force version, of the preceding computer exercise in extracting $\sqrt{(p-1)/2} \pmod{p}$, with p = 103, 1019, 10007, 100003, 1000003, 10000019. Simply compare the runtimes on the particular machine you are working on. Abort any computation if the runtime exceeds five minutes.
Suggestion: You should also make use of Algorithm 2. This not only increases the efficiency of your program, but if your computing platform works in floating point arithmetic (see the note at the beginning of this computer exercise section), it can prevent inaccurate results when intermediate computations involve numbers with too many significant digits.

CHAPTER 13 COMPUTER EXERCISES:

NOTE: The theme of this exercise set will be two-fold. First, the reader will gain some experience in writing programs for search and sort algorithms that involve the manipulation of vectors and lists. After these programs are written, later computer exercises will ask the reader to perform some experiments on randomly generated lists that will provide empirical comparisons of some of these search and sort algorithms. These experiments will nicely motivate the theoretical approach for such comparisons that will be the theme of the next section.

1. (*Program for the Linear Search Algorithm*) (a) Write a program with the following syntax:

```
Ind  = LinearSearch(List,t)
```

The input `List` is a vector of integers, the input `t` is an integer, and the output `Ind` is a nonnegative integer that will be zero if `t` does not appear in `List`, and otherwise will be the smallest index *i* for which `List`(*i*) = `t`. Your program should follow Algorithm 1.
(b) Apply your program to the each of the searches of (ordinary) Exercise 1.

2. (*Program for the Binary Search Algorithm*) (a) Write a program with the following syntax:

```
Ind  = BinarySearch(SortList,t)
```

The input `SortList` is a vector of integers that are sorted from smallest to largest, the input `t` is an integer, and the output `Ind` is a nonnegative integer that will be zero if `t` does not appear in `SortList`, and otherwise will be the smallest index *i* for which `List`(*i*) = `t`.
(b) Apply your program to the each of the searches of Ordinary Exercise 1.

3. (*Program for Random Generation of Sorted Lists*) (a) Write a program with the following syntax:

```
SortList  = SortListGenerator(n,maxGap)
```

The inputs `n` and `maxGap` are positive integers, the output `SortList` is a length *n* vector (list) of positive integers such that `SortList`(1) is a randomly generated positive integer in the range 1 ≤ `SortList`(1) ≤ `maxGap`, and for an index *i* > 1, `SortList`(*i*) is formed by randomly generating a positive integer in the range [1, `maxGap`] and adding it to `SortList`(*i* – 1).
(b) Run your program of Part (a) with the following input settings: `n` = 6, `maxGap` = 10, `n` = 10, `maxGap` = 6.
Suggestion: For Part (a), see the computer exercises of Section 6.2 for information on generating random integers in a specified range.

4. (*Empirical Performance Comparisons Between the Linear and Binary Search Algorithms*)
In this exercise you will compare the run times of the linear search program of Computer Exercise 1 with those of the binary search program of Computer Exercise 2 on some randomly generated sorted lists and targets.
(a) Use the program of Computer Exercise 3 to generate sorted lists starting with size *n* = 2 and `maxgap` = 5, and doubling *n* to 4, 8, 16, …., until the linear search takes more than one minute to execute. Compare the runtimes on each of these inputs of the linear search and the binary search.
(b) Repeat Part (a) but with `maxgap` changed from 5 to 20. Are there any significant differences in the runtimes for each program compared with those of Part (a)?

5. (*Program for the Selection Sort Algorithm*) (a) Write a program with the following syntax:

```
SortList  = SelectionSort(List)
```

The input `List` is a vector of integers, and the output `SortList` is a reordering of the integers in the input into an increasing sequence. The program should operate according to the selection sort Algorithm 3.
(b) Apply your program to each of the sorting problems of Ordinary Exercise 3(a)(c).

6. (*Program for the Bubble Sort Algorithm*) (a) Write a program with the following syntax:

```
SortList  = BubbleSort(List)
```

The input `List` is a vector of integers, and the output `SortList` is a reordering of the integers in the input into an increasing sequence. The program should operate according to the bubble sort Algorithm 4.
(b) Write a program with the following syntax:

```
SortListAll  = BubbleSortShow(List)
```

that operates in exactly the same way as the program in Part (a), except that the output `SortListAll` is a matrix whose *i*th row gives the resulting vector after the *i*th pass through the bubble sort algorithm. Thus the last row of the output matrix should be the same as the output of the program in Part (a), i.e., the sorted list.
(c) Apply your programs of Parts (a) and (b) to the sorting problem of Example 3, and check that the results agree with those of the example.
(d) Apply both programs to each of the sorting problems of Ordinary Exercise 3(a)(c).

7. (*Program for the Quick Sort Algorithm*) (a) Write a program with the following syntax:

```
SortList  = QuickSort(List)
```

The input `List` is a vector of integers, and the output `SortList` is a reordering of the integers of the input into an increasing sequence. The program should operate according to the quick sort Algorithm 5, where the leftmost element in the list is always selected.
(b) Write a program with the following syntax:

```
SortList, IterCount  = QuickSortShow(List)
```

that operates in exactly the same way as the program in Part (a) except it has a second output `IterCount` that gives the number of iterations that were used.
(c) Apply both programs to each of the sorting problems of Ordinary Exercise 3(a)(c).

8. (*Program for the Merge Sort Algorithm*) (a) Write a program with the following syntax:

```
SortList  = Merge(SortListA, SortListB)
```

The inputs `SortListA`, `SortListB` are two sorted lists and the output `SortList` is a reordering of the elements of both lists into a single increasing sequence. The program should operate according to the merging Algorithm 6.
(b) Write a program with the following syntax:

```
SortList = MergeSort(List)
```

The input `List` is a vector of integers, and the output `SortList` is a reordering of the integers in the input into an increasing sequence. The program should operate according to the merge sort Algorithm 7.
(c) Apply both programs to each of the sorting problems of Ordinary Exercise 3(a)(c).

9. (*Program for Random Generation of Unsorted Lists*) (a) Write a program with the following syntax:

```
List  = ListGenerator(n,min,max)
```

The inputs `n` > 0, `min`, and `max` are integers, with `min` ≤ `max`. The output `List` is a length n vector (list) of positive integers that have been randomly generated from the range [`min`, `max`]. Run your program with the following sets of input values: `(n,min,max)` = (6, 1, 10), (10, 0, 1), and (12,1,100).
(b) Write a program with the following syntax:

```
List  = ListGeneratorDistinct(n,min,max)
```

The inputs, output, and functionality are exactly as in Part (b), except that the elements of the output list should be distinct integers (no duplications). One necessary requirement is that `max − min` $\geq n-1$. Run your program with the following sets of input values: `(n,min,max)` = (6, 1, 10), (10, 1, 100), and (12,1,100).
Suggestion: See the computer exercises of Section 6.2 for information on generating random integers in a specified range, with or without duplications.

10. (*Empirical Performance Comparisons Between Sorting Algorithms*)
In this exercise you will compare the runtimes of the various sorting algorithms that were introduced in the section proper, on unsorted lists of increasing size that will be randomly generated using Computer Exercise 9.
(a) Use the program of Computer Exercise 9(a) to generate unsorted lists with the parameter settings `(n,min,max)` = $(k,1,k^2)$, where k runs through the values 2, 4, 8, 16, etc. (doubling at each iteration). For each list run each of the following sorting programs:
(i) `SelectionSort` (Computer Exercise 5)
(ii) `BubbleSort` (Computer Exercise 6(a))
(iii) `QuickSort` (Computer Exercise 7)
(iv) `MergeSort` (Computer Exercise 8(b))
Record the runtimes of each, and continue this process until one of the algorithms takes longer than two minutes to execute. Display the results in graphical form (or with a table).
(b) Weed out the slowest program(s), after doing Part (a), and continue the list generation and run time comparisons with the rest until the next slowest program takes longer than two minutes to execute. Display the results in graphical (or tabular) form.
(c) (*Final Round*) If two programs remain, continue the competition with larger lists.

11. (*Program for Randomized Algorithm that Computes the kth Smallest Element of an Unordered List*) (a) Write a program with the following syntax:

```
element  = Smallest(List,k)
```

The input `List` is a vector of integers, the input `k` is a positive integer at most equal to the length of `List`, and the output `element` is the kth smallest element of `List`. The program should operate according to the randomized Algorithm 8.
(b) Apply your program of Part (a) to find the smallest, second smallest and third smallest elements of the lists of Ordinary Exercise 3(a)(c).

12. (*Empirical Performance Comparisons Between Finding Medians Using the Merge Sort Algorithm Versus Finding them*

Using the Randomized Algorithm 8)
In this exercise, you will perform experiments that compare the run times of the following two algorithms for computing the median of a list of numbers:
(i) Apply the merge sort program (Computer Exercise 8(b)), which is generally the most efficient of the sorting algorithms, to sort the list and then compute the median using the Definition 7.1.
(ii) Apply the randomized Algorithm 8 in conjunction with the strategy from the solution of Example 14.
Use the program of Computer Exercise 9(a) to generate unsorted lists starting with the parameter settings `(n,min,max)` $=(k,1,k^2)$, where k runs through the values 2, 4, 8, 16, etc. (doubling at each iteration). For each list run the above two algorithms to compute the medians of the lists. Record the run times of each, and continue this process until one of the algorithms takes longer than two minutes to execute. Display the results in graphical form (or with a table).

CHAPTER 15 COMPUTER EXERCISES:

1. (*Program for Graphic Sequence Determination Algorithm with Brief Output*) (a) Write a program `Out = GraphSequence(Seq)` that inputs a sequence `Seq` of nonnegative integers, and outputs a truth value `Out` that is either "1" or "0" according as to whether the inputted sequence can (1) or cannot (0) be realized by a simple graph. The program should use Algorithm 1, and, just like the algorithm, should use a while loop structure.
(b) Run your program on the sequences of Example 7 and Exercise for the Reader 8.4, and give the outputs.
(c) Run your program on the sequences of ordinary Exercises 7 and 8, and give the outputs.

2. (*Program for Graphic Sequence Determination Algorithm with Full Output*) (a) Write a program `Out = GraphSequenceShow(Seq)` that inputs a sequence `Seq` of nonnegative integers, and outputs a truth value `Out` that is either "Yes" or "No" according as to whether the inputted sequence can, or cannot be realized by a simple graph. In addition, in either case, the Yes/No output should be followed by a printout of the permuted degree sequences that were produced at the various iterations of the algorithm. The program should use Algorithm 1.
(b) Run your program on the sequences of Example 7 and Example 8, and give the outputs.
(c) Run your program on the sequences of ordinary Exercises 7 and 8, and give the outputs.

3. (*Recursive Program for Graphic Sequence Determination Algorithm with Brief Output*) Although Algorithm 8.1 is recursive in that it repeatedly reduces the given problem to a smaller instance, the algorithm is written with a while loop.
(a) Write a program `Out = GraphSequenceRecursive(Seq)` having the same syntax, input, and outputs as the program of Computer Exercise 1, but that is written in a way so the program calls on itself (rather than using a while loop).
(b) Run your program on the sequences of Example 7 and Example 8 and give the outputs.
(c) Run your program on the sequences of Exercises 7 and 8, and give the outputs.

4. (*Recursive Program for Graphic Sequence Determination Algorithm with Full Output*) (a) Write a program `Out = GraphSequenceShowRecursive(Seq)` having the same syntax, input, and outputs as the program of Computer Exercise 2, but that is written in a way so the program calls on itself (rather than using a while loop).
(b) Run your program on the sequences of Example 7 and Example 8, and give the outputs.
(c) Run your program on the sequences of ordinary Exercises 7 and 8, and give the outputs.

5. (a) Run your program of Computer Exercise 1 on the sequence of threes $(3,3,\cdots,3)$, with length n ranging from 100 to 200. For which values of these n is the sequence graphic? (Compare with the result of ordinary Exercise 9.)
(b) Run your program on the sequence of fours $(4,4,\cdots,4)$, with length n ranging from 100 to 200. For which values of these n is the sequence graphic? (Compare with the result of ordinary Exercise 10.)
(c) Run your program on the sequence of fives $(5,5,\cdots,5)$, with length n ranging from 100 to 200. For which values of these n is the sequence graphic? Make a conjecture as to which values of n the sequence of fives will be graphic (i.e., for which there exists a 5-regular simple graph).

6. Compare the performance of the programs `GraphSequence` of Computer Exercise 1, and `GraphSequenceRecursive` of Computer Exercise 3 by testing both on long sequences of fives $(5,5,\cdots,5)$. Continue doubling the length of the sequence (starting out with length n = 6), until the execution time exceeds five seconds. Indicate how large the sequence becomes (n) when this first occurs for each of the two algorithms.

7. (*Program to Compute Degree Sequences of Graphs*) (a) Write a program with syntax `DegSeq = DegreeSequence(AdjMat)`, with input `AdjMat` being the adjacency matrix of a graph, with respect to some ordering of the vertices, and whose output `DegSeq` is the corresponding sequence of degrees of the vertices.
(b) Run your program on the adjacency matrix of Example 16(a) and Example 8, and give the outputs.
(c) Create adjacency matrices for the simple graphs in Figure 36 (ordering the vertices alphabetically), and then run your

program of Part (a) on each.
(d) Create adjacency matrices for the two general graphs in Figure 38 (ordering the vertices alphabetically), and then run your program of Part (a) on each.

8. (*Program to Compute Degree Sequences of Graphs*) (a) Write a program with syntax `DegSeq = DegreeSequenceI(IncMat)`, with input `IncMat` being the incidence matrix of a graph, with respect to some ordering of the vertices and edges, and whose output `DegSeq` is the corresponding sequence of degrees of the vertices.
(b) Run your program on the incidence matrices of Example 17(a)(b), and give the outputs.
(c) Create incidence matrices for the simple graphs in Figure 36 (ordering the vertices alphabetically, and the edges lexicographically), and then run your program of Part (a) on each.
(d) Create incidence matrices for the two general graphs in Figure 38 (ordering the vertices alphabetically, and the edges any way you wish), and then run your program of Part (a) on each.

9. (*Program to Convert Edge Lists to Adjacency Matrices for Simple Graphs*) (a) Write a program with syntax `AdjMat = EdgeList2AdjMatrix(EdgeList, n)`, with input `EdgeList` being the edge list of a simple graph, where the vertices are labeled with positive integers, ordered with the usual order on numbers. The second output *n* is the number of vertices. The output `AdjMat` is the corresponding adjacency matrix of the graph.
(b) Run your program on the edge list of Figure 27 (where the vertices *a*, *b*, *c*, *d*, *e*, are relabeled as 1, 2, 3, 4, 5).
(c) Write a small program to create an edge list for the 10-path graph P_{10}, and then run the program of Part (a). Give both outputs.
(d) Write a small program to create an edge list for the 10-cycle graph C_{10}, and then run the program of Part (a). Give both outputs.
(e) Will your program or Part (a) work for digraphs? If it does, explain why; otherwise write a related program for digraphs.
Note: In case it is not convenient for your computing platform to implement lists (or if you simply prefer to avoid them), you may simply store edge lists as two-column matrices.

10. (*Program to Convert Edge Lists to Incidence Matrices for Simple Graphs*) (a) Write a program with syntax `IncMat = EdgeList2IncMatrix(EdgeList, n)`, with input `EdgeList` being the edge list of a simple graph, where the vertices are labeled with positive integers, ordered with the usual order on numbers. The second output *n* is the number of vertices. The output `IncMat` is the corresponding incidence matrix of the graph, with the lexicographic ordering on the edges.
(b) Run your program on the edge list of Figure 27 (where the vertices *a*, *b*, *c*, *d*, *e*, are relabeled as 1, 2, 3, 4, 5).
(c) Write a small program to create an edge list for the 10-path graph P_{10}, and then run the program of Part (a). Give both outputs.
(d) Write a small program to create an edge list for the 10-cycle graph C_{10}, and then run the program of Part (a). Give both outputs.
Note: In case it is not convenient for your computing platform to implement lists (or if you simply prefer to avoid them), you may simply store edge lists as two-column matrices. Notice also that the lexicographic ordering of the edges specified for the incidence matrix coincides with the way the edges are ordered in the edge list.

11. (*Program to Convert Adjacency Lists to Adjacency Matrices for Simple Graphs*) (a) Write a program with syntax `AdjMat = AdjList2AdjMatrix(AdjList)`, with input `AdjList` being the adjacency list of a simple graph, where the vertices are labeled with positive integers, ordered with the usual order on numbers. The output `AdjMat` is the corresponding adjacency matrix of the graph.
(b) Run your program on the adjacency list of Figure 27 (where the vertices *a*, *b*, *c*, *d*, *e* are relabeled as 1, 2, 3, 4, 5).
(c) Write a small program to create an adjacency list for the 10-path graph P_{10}, and then run the program of Part (a). Give both outputs.
(d) Write a small program to create an adjacency list for the 10-cycle graph C_{10}, and then run the program of Part (a). Give both outputs.
Note: In case it is not convenient for your computing platform to implement variable length lists (or if you simply prefer to avoid them), you may store adjacency lists as matrices, where the number of columns is the maximum degree of the vertices, and rows corresponding to smaller numbers of vertex neighbors are filled with zeros.

12. (*Program to Convert Adjacency Lists to Incidence Matrices for Simple Graphs*) (a) Write a program with syntax `IncMat = AdjList2IncMatrix(AdjList)`, with input `AdjList` being the adjacency list of a simple graph, where the vertices are labeled with positive integers, ordered with the usual order on numbers. The output `IncMat` is the corresponding incidence matrix of the graph.
(b) Run your program on the adjacency list of Figure 27 (where the vertices *a*, *b*, *c*, *d*, *e* are relabeled as 1, 2, 3, 4, 5).
(c) Write a small program to create an adjacency list for the 10-path graph P_{10}, and then run the program of Part (a). Give both outputs.
(d) Write a small program to create an adjacency list for the 10-cycle graph C_{10}, and then run the program of Part (a). Give both outputs.
Note: In case it is not convenient for your computing platform to implement variable length lists (or if you simply prefer to avoid them), you may store adjacency lists as matrices, where the number of columns is the maximum degree of the

vertices and rows corresponding to smaller number of vertex neighbors are filled with zeros.

13. (*Program to Convert Adjacency Matrices to Incidence Matrices*) (a) Write a program with syntax `IncMat = AdjMatrix2IncMatrixSimple(AdjMat)`, with input `AdjMat` being the adjacency matrix of a simple graph, where the vertices are labeled with positive integers, ordered with the usual order on numbers. The output `IncMat` is the corresponding incidence matrix of the graph, where the edges are labeled lexicographically.
(b) Run your program on the adjacency matrix of Example 16(b).
(c) Write a small program to create an adjacency matrix for the 10-path graph P_{10}, and then run the program of Part (a). Give both outputs.
(d) Write a small program to create an adjacency matrix for the 10-cycle graph C_{10}, and then run the program of Part (a). Give both outputs.
(e) Modify your program of Part (a), into a new program with syntax `IncMat = AdjMatrix2IncMatrix(AdjMat)` that is able to handle adjacency matrices of general graphs.
(f) Run your program of Part (e) on the incidence matrix of the solution of Part (a) of Example 16. Give the output.
(g) Run your program of Part (e) on the incidence matrices for the two general graphs of Figure 38 (where the vertices are ordered alphabetically). Give the output.
Note: For Part (e), the edges are ordered lexicographically so multiple edges will be adjacent and their columns in the incidence matrix will be identical. Thus, the way in which multiple edges are ordered amongst themselves is not relevant.

14. (*Program to Convert Incidence Matrices to Adjacency Matrices*) (a) Write a program with syntax `AdjMat = IncMatrix2AdjMatrix(IncMat)`, with input `IncMat` being the incidence matrix of a graph, where the vertices are labeled with positive integers, ordered with the usual order on numbers, and the edges lexicographically. The output `AdjMat` is the corresponding adjacency matrix of the graph.
(b) Run your program on the incidence matrix of Example 14(c).
(c) Run your program on the incidence matrices of ordinary Exercise 28(b).

15. (*Program to Compute Second Degree Sequences of Simple Graphs*) (a) Write a program with syntax `SecDegSeq = SecondDegreeSequence(AdjMat)`, with input `AdjMat` being the adjacency matrix of a simple graph, with respect to some ordering of the vertices, and whose output `DegSeq`, is the corresponding sequence of second degrees of the vertices. See ordinary Exercise 52 for the relevant definition.
(b) Run your program on adjacency matrices for the three eight-vertex simple graphs of ordinary Exercise 22. Do the outputs tell you anything about whether the graphs are isomorphic?
(b) Run your program on adjacency matrices for the four seven-vertex simple graphs of ordinary Exercise 23. Do the outputs tell you anything about whether the graphs are isomorphic?
(c) Write a small program to create an adjacency matrix for the n-path graph P_n. Use this program, along with the program of Part (a) to compute the second degree sequences of the n-path graphs for n = 10, 11, and 12. Do you observe any pattern?
(d) Write a small program to create an adjacency matrix for the n-cycle graph C_n. Use this program, along with the program of Part (a) to compute the second degree sequences of the n-cycle graphs for n = 10, 11, and 12. Do you observe any pattern?

16. (*Brute-Force Isomorphism Checker for Simple Graphs*) (a) Write a program
`Out, Perm = IsomorphismChecker(AdjMat1, AdjMat2)`
that inputs a pair of adjacency matrices `AdjMat1, AdjMat2` that correspond to two simple graphs having the same number of vertices (with some specified ordering of their vertices), and outputs a truth value `Out` that is either "1" or "0" according as to whether the inputted graphs are isomorphic (1 = True, 0 = False). The second output variable `Perm` will be a vector representing a permutation corresponding an isomorphism, in case the graphs are isomorphic; otherwise it will be the single number 0. The program should go through all vertex permutations on the first matrix to check to see if it can be transformed into the second matrix. In case an isomorphism is found, the program exits and outputs also the (first) vertex permutation that rendered the matrices equal (and hence produces a graph isomorphism).
(b) Apply your program to the isomorphism problem of ordinary Exercise 20.
(c) Apply your program to the isomorphism problem of ordinary Exercise 24.
(d) Apply your program to the isomorphism problem of ordinary Exercise 25.

17. (*Brute-Force Isomorphism Checker for Simple Digraphs*) (a) Will your program or Part (a) of the previous computer exercise work for digraphs? If it does, explain why; otherwise write a related program for digraphs.
(b) Apply your program to the isomorphism problem of ordinary Exercise 38.
(c) Apply your program to the isomorphism problem of ordinary Exercise 39.
(d) Apply your program to the isomorphism problem of ordinary Exercise 40.

CHAPTER 16 COMPUTER EXERCISES:

1. (*Program for Algorithm 1*) (a) Write a program for computing the diameter of a network (i.e., either a graph or a digraph) based on Algorithm 1. The syntax should be as follows:

```
diam = NetworkDiameterAlg8_2(A)
```

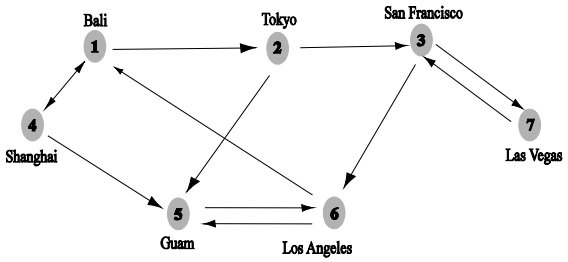
where the input `A` is the adjacency matrix for the network, and the output `diam` is the diameter of the network.
(b) Check your program with the results obtained for the networks of Examples 9 and 10.

2. (*Program for Computing Vertex Eccentricities in a Graph*) (a) Write a program for computing all of the vertex eccentricities in a network (i.e., either a graph or a digraph) that is based on computing powers of the adjacency matrix. The syntax should be as follows:

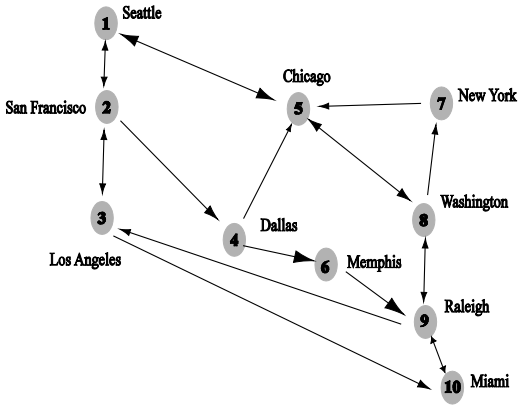
```
EccVec = NetworkEccentricities(A)
```

where the input `A` is the adjacency matrix for the network, and the output `EccVec` is a vector of the corresponding vertex eccentricities.
(b) Check your program on the networks of Examples 11 and 10.
Suggestion: The main idea for the program is contained in the solution of Example 10. But the program should be structured so that internally it computes the sequence $I + A + A^2$, $I + A + A^2 + A^3, \ldots$ only once, reading off any new eccentricities of vertices discovered as each new power of A is added.

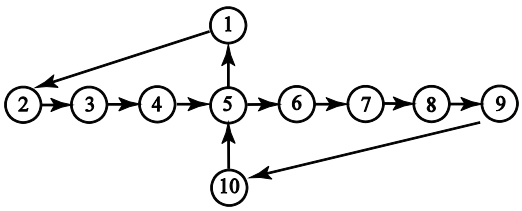
3. (*Some Small Network Questions for the Programs of Computer Exercises 1 and 2*) Run your programs of Computer Exercises 1 and 2 to compute the diameters and vertex eccentricities for each of the following networks:
(a) The seven-city network shown below:



(b) The 10-city network shown below:



(c) The 10-node network shown below:



(d) The 23-city network of Figure 8.

NOTE: (***Diameter Reduction in Networks***) We again use the familiar setting of shipping networks to motivate the topic of diameter reduction in a network. Suppose that the manager of a certain air shipping network feels that the diameter of the network is too large (for example, think of the 23-city network of Figure 8, which has diameter 8), and wants to consider the feasibility of adding some new flights in order to reduce the

diameter. Suppose first that the manager would like to know how much of a reduction is possible with the addition of a single new route (i.e., a single new arrow in the network diagram), and furthermore, wishes to know all of the new routes that would yield this optimal reduction.[5] Since adding one new edge to a digraph network corresponds to changing the corresponding off diagonal entry in the adjacency matrix from 0 to 1 (for a graph network two entries in the matrix need to be changed corresponding to the two directions between the cities), the brute-force approach would be to simply use Algorithm 1/Computer Exercise 1 (or whatever other program we have for computing network diameters) to compute the diameters of all of the different modified adjacency matrices that arise in this way and keep track of the routes that give rise to the lowest diameters. The following two computer exercises will put these ideas into practice.

4. (*Diameter Reduction in a Digraph Network—One New Directed Edge*)
   (a) Write a program with the following syntax:

   `OldDiam,NewDiam,Edges = DiamReduction1NewArrow(A)`

   where the input `A` is the adjacency matrix for a digraph network, and the outputs are as follows: `OldDiam` is the diameter of the original network, `NewDiam` is the lowest diameter of a new digraph obtained from the original by adding one new (directed) edge, and `Edges` is a two-column matrix whose rows are all of the directed edges that will realize this diameter reduction. In the case that `NewDiam = OldDiam` (i.e., no diameter reduction is possible), `Edges` should be the empty matrix.
   (b) Run your program on the networks of Examples 10 and 11.
   (c) Run your program on each of the networks of Computer Exercise 3.

5. (*Diameter Reduction in a Digraph Network—Two New Directed Edges*)
   (a) Write a program with the following syntax:

   `OldDiam,NewDiam,Edges = DiamReduction2NewArrows(A)`

   where the input `A` is the adjacency matrix for a digraph network, and the outputs are as follows: `OldDiam` is the diameter of the original network, `NewDiam` is the lowest diameter of a new digraph obtained from the original by adding two new (directed) edges, and `Edges` is a four-column matrix whose rows are all of pairs of directed edges that will realize this diameter reduction. In that case that `NewDiam = OldDiam` (i.e., no diameter reduction is possible), `Edges` should be the empty matrix.
   (b) Run your program on the networks of Examples 9 and 12.
   (c) Run your program on each of the networks of Computer Exercise 3.
   (d) Does each pair of edges that realizes a maximum reduction of diameter with two new directed edges necessarily contain an edge corresponding to a maximum reduction of diameter using one new directed edge? Explain your answer.

NOTE: (***Random Graphs***) Randomly generated graphs (or digraphs) are often useful for an assortment of applications and hypothesis testing. The concept of how to randomly generate a graph on n vertices is quite simple: There are $C(n,2)$ possible edges, corresponding to pairs of distinct vertices. For each of these possible edges, we flip a fair coin; if it turns out heads, we include the edge; otherwise we do not. This process is easily programmed on a computer using adjacency matrices and random number generator. Because of its importance, we highlight the procedure as the following algorithms, which have a more general feature of being able to specify the target edge density (corresponding to changing the fair coin to a biased one in the above explanation).

ALGORITHM 2A: Random Generation of a Simple Digraph with Specified Edge Density:

Input: A positive integer n representing the number of vertices of the random digraph to be generated, and a positive number p, $0 < p < 1$, representing the edge density.
Output: The n by n adjacency matrix A of a simple digraph that is randomly generated, where the directed edges between pairs of vertices are independently selected with probability p.

Step 1: Initialize the matrix A as the n by n matrix of zeros.
Step 2: (*Random Edge Selection*) Proceed through each of the non-diagonal entries of A, and with probability p change the entry from 0 to 1. This is accomplished by the following more precise nested for loop:

```
FOR i = 1 TO n (i is the row index)
        FOR j = 1 TO n (j is the column index)
            Set U = rand (i.e., generate a random real number in (0,1) from the uniform distribution)
          IF U < p (the probability of this event is p) AND  i ≠ j (don't change diagonal entries)
              Set A(i,j) = 1 (i.e., we add a directed edge from vertex #i to vertex #j)
          END IF
        END j FOR
END i FOR
```

The corresponding algorithm for graphs is similar, we just need to maintain $A(i,j) = A(j,i)$.

ALGORITHM 2B: Random Generation of a Simple Graph with Specified Edge Density:

[5] Although from a mathematical perspective, the existence of one such new route is usually sufficient knowledge, from a business perspective it is often necessary to know all options. Indeed, establishing a new route between a pair of cities requires negotiations and cost, and some options may be more feasible than others.

Input: A positive integer n representing the number of vertices of the random graph to be generated, and a positive number p, $0 < p < 1$, representing the edge density.
Output: The n by n adjacency matrix A of a simple graph that is randomly generated, where the edges between pairs of vertices are independently selected with probability p.

Step 1: Initialize the matrix A as the n by n matrix of zeros.
Step 2: (*Random Edge Selection*) Proceed through each of the above-diagonal entries of A, and with probability p change the entry (and the corresponding below-diagonal entry) from 0 to 1. This is accomplished by the following more precise nested for loop:

```
FOR i = 1 TO n (i is the row index)
        FOR j = i + 1 TO n (j is the column index)
            Set U = rand (i.e., generate a random real number in (0,1) from the uniform distribution)
          IF U < p (the probability of this event is p)
              Set A(i,j) = 1 and A(j,i) = 1 (i.e., we add an edge between vertex #i to vertex #j)
           END IF
        END j FOR
END i FOR
```

6. (*Program for Random Generation of Simple Graphs with Specified Edge Density*)
   (a) Write a program with the following syntax:

   `A = RandomSimpleGraph(n,p)`

   where the input `n` is a positive integer corresponding to the number of vertices of the random simple graph that will be generated, the input `p` is the edge selection probability (i.e., the edge density), and output `A` is an adjacency matrix produced by an application Algorithm 2B.
   (b) Run your program with parameter settings `n` = 100, and `p` = 0.15, and count the number of edges in the resulting graph. Repeat 20 times.

7. (*Program for Random Generation of Simple Digraphs with Specified Edge Density*)
   (a) Write a program with the following syntax:

   `A = RandomSimpleDigraph(n,p)`

   where the input `n` is a positive integer corresponding to the number of vertices of the random simple digraph that will be generated, the input `p` is the edge selection probability (i.e., the edge density), and output `A` is an adjacency matrix produced by an application Algorithm 2A.
   (b) Run your program with parameter settings `n` = 100 and `p` = 0.15, and count the number of directed edges in the resulting graph. Repeat 20 times.

8. (*Estimating the Probability that a Randomly Generated Simple Graph is Connected*)
   (a) Use your program of Computer Exercise 6 to randomly generate 100 10-vertex simple graphs with edge density p = 0.5, and for each of them use your program of Computer Exercise 1 to check whether the graph is connected (i.e., the diameter is finite). Use these results to estimate the probability that a randomly generated 10-vertex simple graph (with edge density p = 0.5) is connected.
   (b) Repeat Part (a), but with the number of vertices changed to 20.
   (c) Repeat Part (a), but with the edge density changed to p = 0.10.
   (d) Repeat Part (a), but with the edge density changed to p = 0.10, and the number of vertices changed to 20.
   (e) It is clear that for a fixed number n of vertices, the probability that a randomly generated graph on n vertices with edge density p is connected increases with p. In the extreme cases when p = 0 (no edges) this probability is zero, and p = 1 (complete graph) this probability is one. Do some more computer experiments to estimate the value of p that will make this probability equal to 1/2 in the case of a 10-vertex graph.

9. (*Estimating the Probability that a Randomly Generated Simple Digraph is Connected*)
   (a) Use your program of Computer Exercise 7 to randomly generate 100 10-vertex simple digraphs with edge density p = 0.5, and for each of them use your program of Computer Exercise 1 to check whether the digraph is connected (i.e., the diameter is finite). Use these results to estimate the probability that a randomly generated 10-vertex simple digraph (with edge density p = 0.5) is (strongly) connected.
   (b) Repeat Part (a), but with the number of vertices changed to 20.
   (c) Repeat Part (a), but with the edge density changed to p = 0.10.
   (d) Repeat Part (a), but with the edge density changed to p = 0.10, and the number of vertices changed to 20.
   (e) It is clear that for a fixed number n of vertices, the probability that a randomly generated digraph on n vertices with edge density p is connected increases with p. In the extreme cases when p = 0 (no edges) this probability is zero, and p = 1 (complete digraph) this probability is one. Do some more computer experiments to estimate the value of p that will make this probability equal to 1/2 in the case of a 10-vertex digraph.

10. (*Adjacency Matrix-Based Program for Determination of the Components of a Graph*)
    (a) Write a program for computing all of the components of a simple graph. Base the program on computing powers of the adjacency matrix. The syntax should be as follows:

    `CompVec = Components(A)`

    where the input `A` is the adjacency matrix for the network, and the output `CompVec` is a vector of positive integers that are associated with the vertices (in the order corresponding to the adjacency matrix) and defined as follows: vertex #1 is

assigned the (component) number 1, as are all other vertices in the same component as vertex #1. The next vertex in the graph not in the same component as vertex #1 (i.e., in case the graph is not connected) will be assigned (component) number 2, as will be all other vertices that are also in this component. The scheme continues in this fashion.
(b) Check your program against the results of Parts (b) and (c) of ordinary Exercise 41.
(c) Use the program of Part (a) in conjunction with the `RandomSimpleGraph` program of Computer Exercise 6 to estimate the expected number of components of a random 20-vertex simple graph with edge density $p = 0.03$. Generate at least 100 graphs.
Note: For Part (a), the solution to ordinary Exercise 41 describes the relevant algorithm.

CHAPTER 17 COMPUTER EXERCISES:

NOTE: We will assume throughout that vertex labels of trees are positive integers. This removes any technical issues in comparing two vertices since most computing systems have built-in functionality to compare two integers (i.e., given two different integers a, b, determine whether $a < b$ or $a > b$). Any examples of the section referred to in this set whose vertex labels are not integers should be given integer labels using a natural scheme, such as the following: $a = 1, b = 2, \ldots, z = 26$.[6] This set of computer exercises will develop some programs for rooted trees. The basic data structures will be the parent list and the children list, which are described in Example 19. In particular, the parent list will be a two-column matrix where the first column lists the vertices (in order) and the second column their corresponding parents. The children list will always be an $m + 1$ column matrix where m is the maximum number of children (i.e., the smallest value of m such that the rooted tree is an m-ary tree). The first column lists all the vertices in order (but putting the root first), and the remaining columns list the corresponding children (in order) with some convenient label (such as "0" or "null") used to fill empty slots in cases where there are less than m children; we will use 0s to fill such slots.

1. (*Children List to Parent List*) (a) Write a program that will input a children list matrix for a rooted tree and will output the corresponding parent list matrix. The syntax should be as follows:

```
ParentList = ChildrenList2ParentList(ChildrenList)
```

The input `ChildrenList` should be an $m + 1$ column matrix (of integers), and the output `ParentList` a two-column matrix as described in the preceding note.
(b) Apply your program to the binary tree of Figure 17.
(c) Apply your program to the binary tree of Figure 20.
(d) Apply your program to the 4-ary tree of Figure 16.

2. (*Parent List to Depth List*) (a) Write a program that will input a parent list matrix for a rooted tree and will output the corresponding *depth list* two-column matrix, whose first column lists the vertices (in the same order as the inputted parent list matrix) and whose corresponding second column entries give the nonnegative integer depths of the vertices. The syntax should be as follows:

```
DepthList = ParentList2DepthList(ParentList)
```

(b) Apply your program to the binary tree of Figure 17.
(c) Apply your program to the binary tree of Figure 20.
(d) Apply your program to the 4-ary tree of Figure 16.
Suggestion: Rather than trace the path back to the root for each vertex, it is much more efficient to progress down through the successive generations of vertices, assigning the common depth to each generation.

3. (*Program to Compute All Descendants of an Inputted Vertex in a Rooted Tree*) (a) Write a program that determines all descendants of an inputted vertex in an inputted rooted tree. The syntax should be as follows:

```
DescendantList = DescendantFinder(ChildrenList, vertex)
```

The input `ChildrenList` should be as described in the note at the beginning of this computer exercise set, and the second input `vertex` should be an (actual) vertex number. The output will be a vector (possibly empty) of all vertex labels that are descendants of the inputted vertex.
(b) Apply your program to the binary tree of Figure 17 with the root vertex, and with a couple of vertices of different depths.
(c) Apply your program to the binary tree of Figure 20 with the root vertex, and with a couple of vertices of different depths.
(d) Apply your program to the 4-ary tree of Figure 16 with the root vertex, and with a couple of vertices of different depths.

4. (*Program to Compute All Ancestors of an Inputted Vertex in a Rooted Tree*) (a) Write a program that determines all descendants of an inputted vertex in an inputted rooted tree. The syntax should be as follows:

```
AncestorList = AncestorFinder(ChildrenList, vertex)
```

[6] Readers who have more experience with programming may wish to extend the functionality of some of the programs to include the comparisons of strings of letters using lexicographic order.

The input `ChildrenList` should be as described in the note at the beginning of this computer exercise set, and the second input `vertex` should be a vertex number. The output will be a vector (possibly empty) of all vertex numbers that are ancestors of the inputted vertex.
(b) Apply your program to the binary tree of Figure 17 with the root vertex, and with a couple of vertices of different depths.
(c) Apply your program to the binary tree of Figure 20 with the root vertex, and with a couple of vertices of different depths.
(d) Apply your program to the 4-ary tree of Figure 16 with the root vertex, and with a couple of vertices of different depths.

5. (*Rooted Tree to Children Subtrees*) (a) Write a program that will input a children list matrix for a rooted tree and will output children list matrices for each of the children subtrees of the root. The syntax should be as follows:

```
ChildChildrenLists = ChildTrees(ChildrenList)
```

The input `ChildrenList` corresponds to the inputted tree, the output `ChildChildren Lists` is an ordered list of *s* matrices each being a children list for one of the child subtrees of the root of the inputted tree. Here *s* denotes the number of children of the root of the inputted tree; see Figure 10.
(b) Apply your program to the binary tree of Figure 17.
(c) Apply your program to the binary tree of Figure 20.
(d) Apply your program to the 4-ary tree of Figure 16.
Suggestion: The most straightforward data structure for the output variable in this program is a three-dimensional array, where the third dimension is the index corresponding to the child subtree represented by each two-dimensional matrix. An ordered list of matrices is really a three-dimensional array. Here the first two dimensions of the three-dimensional array are the common dimensions of the children list matrices, and the third dimension determines the number of matrices in the list. In order to create a well defined array, use the same number of rows for each of the outputted children tree matrices. The number of rows will be the depth of the original tree, which can be computed by calling on the programs of Computer Exercises 1 and 2. Unallocated rows in outputted child tree matrices can be filled with zeros.

6. (*Preorder Traversal of a Rooted Tree-Recursive Program*) (a) Write a program that will input a children list matrix for a rooted tree and will output a vector containing the preorder list of the vertices. The program should be recursive and based on Algorithm 1. The syntax should be as follows:

```
PreOrder = PreOrderAlg(ChildrenList)
```

(b) Apply your program to the binary tree of Figure 17.
(c) Apply your program to the binary tree of Figure 20.
(d) Apply your program to the 4-ary tree of Figure 16.
Suggestion: One possible strategy to create a program that captures Algorithm 1 is to have the program `PreOrderAlg` consist of a single line, which calls on a recursive program that requires two inputs:

```
PreOrder = PreOrderRecursiveAlg(ChildrenList,List)
```

where the second input variable represents a partial list that is being built up in going from one iteration to the next. The former program should call on the latter with the second input variable being an *empty list*, since the first time we enter into the recursive algorithm the list is empty. Thus, the single line of code for the first program will look something like: `PreOrder = PreOrderRecursiveAlg(ChildrenList, []).` A possible pseudocode for the latter recursive program is as follows:

```
IF ChildrenList has nonzero entries (i.e., the tree is nonempty)
   Append the first entry (the root) to List
END (IF)
IF ChildrenList has more than one nonzero entry (i.e., the root
                                                 has children)
        Set s = the number of children
        Set ChildChildrenLists = ChildTrees(ChildrenList)
        (i.e., apply the program of Computer Exercise 5)
        FOR i = 1 TO s
             Set CMi = ith Child Tree Children List
             (i.e., the ith matrix of ChildChildrenLists)
             Set PreOrder = PreOrderRecursiveAlg(CMi, PreOrder)
        END (FOR)
END (IF)
```

7. (*Inorder Traversal of a Rooted Tree-Recursive Program*) (a) Write a program that will input a child list matrix for a rooted tree and will output a vector containing the inorder list of the vertices. The program should be recursive and based on Algorithm 2. The syntax should be as follows:

```
InOrder = InOrderRecursiveAlg(ChildrenList)
```

(b) Apply your program to the binary tree of Figure 17.
(c) Apply your program to the binary tree of Figure 20.
(d) Apply your program to the 4-ary tree of Figure 16.

Suggestion: Use the ideas of the suggestion for Computer Exercise 6.

8. (*Postorder Traversal of a Rooted Tree-Recursive Program*) (a) Write a program that will input a child list matrix for a rooted tree and will output a vector containing the postorder list of the vertices. The program should be recursive and based on Algorithm 3. The syntax should be as follows:

```
PostOrder = PostOrderRecursiveAlg(ChildrenList)
```

(b) Apply your program to the binary tree of Figure 17.
(c) Apply your program to the binary tree of Figure 20.
(d) Apply your program to the 4-ary tree of Figure 16.
Suggestion: Use the ideas of the suggestion for Computer Exercise 6.

9. (*Recursive Search and/or Appending in a Binary Search Tree*) (a) Write a program that will input a child list matrix `ChildrenList` for a rooted tree and a label, `LABEL`, and will output either the row index of the child list matrix in which the label appears as the first column entry (i.e., the label was located in the tree), or will ouput a new `ChildrenList` matrix, with one additional row and corresponding to the proper insertion of a new vertex in the tree that carries the inputted label. The program should follow Algorithm 5, and the syntax should be as follows

```
Output = BinarySearchTreeFinderInserter(ChildrenList, LABEL)
```

(b) Apply your program to the binary search tree creation problem of ordinary Exercise 19(b).
Suggestion: Use the ideas of the suggestion for Computer Exercise 6.