Chapter 13
Recursion and Advanced Algorithms

Algorithms for searching and sorting data will be covered in this chapter. Recursion will be used in the implementation of some of these algorithms.

Selection Sort

Sorting is the process of putting items in a designated order, either from low to high or high to low. The *selection sort* algorithm starts by finding the lowest item in a list and swapping it with the first. Next, the lowest item among items 2 through the last is found and swapped with item 2, and then the lowest item among items 3 through the last is swapped with item 3. This process is continued until the last item is reached, at which point all the items are sorted.

The selection sort algorithm compares an element to the items in the array after the element. This algorithm can be implemented with nested `for` loops. The outer loop controls which element to compare and the inner `for` loop iterates through the array after the element (the subarray). The selection sort pseudocode for sorting an array of items from low to high appears like:

```
for (arrayIndex = 0 to numItems-1)
  for (subarrayIndex = arrayIndex to numItems-1)
    if (items[subarrayIndex] < items[arrayIndex]) {
      swap items[arrayIndex] and items[arrayIndex]
    }
  }
}
```

The Sorts class on the next page implements a selectionSort() method:

```
public class Sorts {

  /**
   * Sorts an array of data from low to high
   * pre: none
   * post: items has been sorted from low to high
   */
  public static void selectionSort(int[] items) {

    for (int index=0; index<items.length; index++) {
      for (int subIndex=index; subIndex<items.length; subIndex++) {
        if (items[subIndex] < items[index]) {
          int temp = items[index];
          items[index] = items[subIndex];
          items[subIndex] = temp;
        }
      }
    }
  }
}
```

The TestSorts application generates an array of integers and then calls selectionSort() to sort the array elements:

```
import java.util.Scanner;
import java.util.Random;

public class TestSorts {

  public static void displayArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + "  ");
    }
    System.out.println("\n");
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int numItems;
    int[] test;
    Random rand = new Random();

    System.out.print("Enter number of elements: ");
    numItems = input.nextInt();

    /* populate array with random integers */
    test = new int[numItems];
    for (int i = 0; i < test.length; i++) {
      test[i] = rand.nextInt(100);
    }
    System.out.println("Unsorted:");
    displayArray(test);

    Sorts.selectionSort(test);

    System.out.println("Sorted:");
    displayArray(test);
  }
}
```

The TestSorts application produces output similar to:

```
Enter number of elements: 10
Unsorted:
91  36  4  65  45  67  87  93  24  48

Sorted:
4  24  36  45  48  65  67  87  91  93
```

Sorting Objects

equals(), compareTo()

Relational operators, such as > and <, cannot be used to compare objects. Objects use methods of their class to determine if one object is greater than, less than, or equal to another. The equals() method in a class is used to determine equality. For determining order, the compareTo() method is used.

Comparable interface

Objects that are to be sorted must have a class that implements the Comparable interface. The String, Double, and Integer clases implement the Comparable interface. The Circle class in Chapter 9 also implements the Comparable interface.

polymorphism

An interface cannot be used to instantiate a class. However, an interface can be used as a data type. An interface data type can reference any class that implements it. This polymorphic behavior makes it possible to implement a generic sort that works with any list of objects that implement the Comparable interface.

The Sorts class has been modified to include an overloaded SelectionSort() method, which has a Comparable array parameter:

```
/**
 * Sorts an array of objects from low to high
 * pre: none
 * post: Objects have been sorted from low to high
 */
public static void selectionSort(Comparable[] items) {

  for (int index = 0; index < items.length; index++) {
    for (int subIndex=index; subIndex<items.length; subIndex++) {
      if (items[subIndex].compareTo(items[index]) < 0) {
        Comparable temp = items[index];
        items[index] = items[subIndex];
        items[subIndex] = temp;
      }
    }
  }
}
```

The TestSorts application, on the next page, has been modified to sort an array of Circle objects:

```
import java.util.Scanner;
import java.util.Random;

public class TestSorts {

  public static void displayArray(Circle[] array) {
    for (int i = 0; i < array.length; i++) {
      System.out.println(array[i] + "  ");
    }
    System.out.println("\n");
  }


  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int numObjects;
    Circle[] test;
    Random rand = new Random();

    System.out.print("Enter number of objects: ");
    numObjects = input.nextInt();
    input.close();

    /* populate array with Circle objects of varying radii */
    test = new Circle[numObjects];
    for (int i = 0; i < test.length; i++) {
      test[i] = new Circle(rand.nextInt(10));
    }
    System.out.println("Unsorted:");
    displayArray(test);

    Sorts.selectionSort(test);

    System.out.println("Sorted:");
    displayArray(test);
  }
}
```

The TestSorts application produces output similar to:

```
Enter number of objects: 5
Unsorted:
Circle has radius 5.0
Circle has radius 6.0
Circle has radius 5.0
Circle has radius 3.0
Circle has radius 8.0


Sorted:
Circle has radius 3.0
Circle has radius 5.0
Circle has radius 5.0
Circle has radius 6.0
Circle has radius 8.0
```

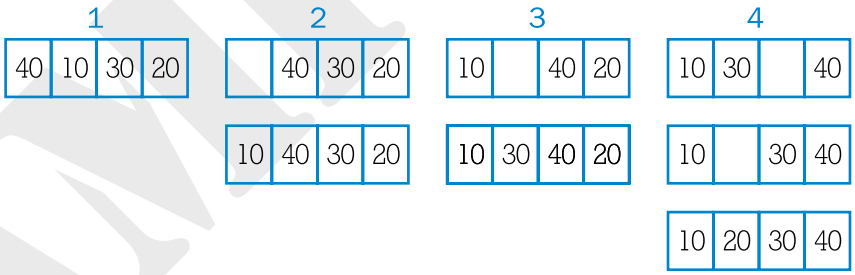
Review: ArrayListSort

Create an ArrayListSort application that implements a selection sort on an ArrayList object. Test the sort with an ArrayList containing Double objects.

Insertion Sort

More efficient than the selection sort algorithm is the insertion sort algorithm. An *insertion sort* starts by sorting the first two items in a list. This sort is performed by shifting the first item into the second spot if the second item belongs in the first spot. Next, the third item is properly inserted within the first three items by again shifting items into their appropriate position to make room for the moved item. This process is repeated for the remaining elements.

The insertion sort is illustrated below with an array containing four elements. Step 1 shows the original list, which contains items 40, 10, 30, and 20. Step 2 shows that 40 is shifted to make room for the second item, 10. Next, 30 compared to the value in the previous position (40), 40 is shifted into position 3, 30 is then compared to the value in the previous position (10), and then 30 is placed at position 2. This process repeats for the remaining items.



Based on the algorithm, the insertion sort pseudocode for an array of integers is:

```
for (index = 1 To array.length - 1) {
  temp = array[index]
  previousIndex = index - 1
  while (array[previousIndex] > temp && previousIndex > 0) {
    shift array[previousIndex] up one element
    previousIndex = previousIndex - 1
  }
  if (array[previousIndex] > temp) {
    swap the two elements
  } else {
    insert element at appropriate location
  }
}
```

The Sorts class has been modified to include an insertionSort() method:

```
/**
 * Sorts an array of integer from low to high
 * pre: none
 * post: Integers have been sorted from low to high
 */
public static void insertionSort(int[] items) {
  int temp, previousIndex;

  for (int index = 1; index < items.length; index++) {
    temp = items[index];
    previousIndex = index - 1;
    while ((items[previousIndex] > temp) && (previousIndex > 0)) {
      items[previousIndex + 1] = items[previousIndex];
      previousIndex -= 1;       //decrease index to compare current
    }                           //item with next previous item
    if (items[previousIndex] > temp) {
      /* shift item in first element up into next element */
      items[previousIndex + 1] = items[previousIndex];
      /* place current item at index 0 (first element) */
      items[previousIndex] = temp;
    } else {
      /* place current item at index ahead of previous item */
      items[previousIndex + 1] = temp;
    }
  }
}
```

The TestSorts application has been modified to use the insertionSort() method to sort an array of integers:

```
import java.util.Scanner;
import java.util.Random;

public class TestSorts {

  public static void displayArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + "  ");
    }
    System.out.println("\n");
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int numItems;
    int[] test;
    Random rand = new Random();

    System.out.print("Enter number of elements: ");
    numItems = input.nextInt();

    /* populate array with random integers */
    test = new int[numItems];
    for (int i = 0; i < test.length; i++) {
      test[i] = rand.nextInt(100);
    }
    System.out.println("Unsorted:");
    displayArray(test);

    Sorts.insertionSort(test);
```

```
        System.out.println("Sorted:");
        displayArray(test);
    }
}
```

The TestSorts application produces output similar to:

```
Enter number of elements: 10
Unsorted:
65  52  75  10  44  86  37  14  13  39

Sorted:
10  13  14  37  39  44  52  65  75  86
```

Review: ObjectsInsertionSort

Create an ObjectsInsertionSort application that implements an insertion sort on an array of objects. Test the sort on an array of String objects.

Recursion

recursive call

A method can call itself. This process is called *recursion* and the calls are referred to as *recursive calls*. The RecursiveDemo application contains a method that calls itself:

```
public class RecursiveDemo {

    public static void showRecursion(int num) {
        System.out.println("Entering method.  num = " + num);
        if (num > 1) {
            showRecursion(num - 1);
        }
        System.out.println("Leaving method.  num = "  + num);
    }


    public static void main(String[] args) {
        showRecursion(2);
    }
}
```

The call `showRecursion(2)` from the main() method is the initial call. In the showRecursion() method, a call is made to itself passing `num - 1`. When showRecursion is called with num equal to 1, the `if` is skipped and the remaining statement in the method is executed. At this point the stack of calls made before num was 1 are executed in the reverse order they were made, with each call executing the statement in the method after the recursive call (after the `if`).

The RecursiveDemo produces the output shown on the next page:

```
Entering method. num = 2
Entering method. num = 1
Leaving method. num = 1
Leaving method. num = 2
```

Recursion is a programming technique that can be used whenever a problem can be solved by solving one or more smaller versions of the same problem and combining the results. The recursive calls solve the smaller problems.

One problem that has a recursive solution is raising a number to a power. For example, 2^4 can be thought of as:

$2^4 = 2 * 2^3$ which can be thought of as
$2^3 = 2 * 2^2$ which can be thought of as
$2^2 = 2 * 2^1$ which can be thought of as
$2^1 = 2 * 2^0$ which can be thought of as
1

In each case, the power problem is reduced to a smaller power problem. A more general solution is:

$x^n = x * x^{n-1}$

infinite recursion
base case

However, carefully analyzing this solution shows that there is no stopping point. For example, 2^3 would be $2 * 2^2$, which would return $2 * 2^1$, which returns $2 * 2^0$, which returns $2 * 2^{-1}$, and so on. This solution would cause *infinite recursion*. To prevent this, a recursive solution must have a *base case* that requires no recursion. For this solution, when the power is 0, 1 should be returned.

The intPower() method in the Power application implements a recursive solution for calculating an `int` raised to an `int` power to return an `int` (as opposed to Math.pow(), which returns a `double`):

```
public class Power {

  /**
   * Returns num to the power power.
   * pre: num and power are not 0.
   * post: num to the power power has been returned.
   */
  public static int intPower(int num, int power) {
    int result;
    if (power == 0) {
      result = 1;
    } else {
      result = num * intPower(num, power-1);
    }
    return(result);
  }


  public static void main(String[] args) {
    int x = intPower(2, 5);
    System.out.println(x);
  }
}
```

The Power application produces the output:

```
32
```

Review: RecursiveFactorial

Create a RecursiveFactorial application that returns the factorial of an integer. The factorial of a number is the product of all positive integers from 1 to the number. For example, 5! = 5*4*3*2*1. Computing 5! could be thought of as 5*4! or more generally, n*(n–1)!. By definition, 0! is equal to 1. Compare your recursive solution to the nonrecursive solution created in the Factorial Review completed in Chapter 6.

Mergesort

TIP Measuring an algorithm's efficiency is discussed later in this chapter.

The selection sort is simple, but inefficient, especially for large arrays. Imagine using the selection sort process by hand for a pile of 1000 index cards. Searching through the cards for the lowest item would take a long time, but more importantly, after each search the remaining cards must be searched again! Each card ends up being examined about 500 times.

The *mergesort* algorithm takes a "divide-and-conquer" approach to sorting. Imagine the 1000 cards being divided into two piles of 500. Each pile could then be sorted (a simpler problem) and the two sorted piles could be combined (merged) into a single ordered pile. To further simplify the sorting, each subpile could be divided and sorted, and so on. This algorithm is best implemented recursively.

The mergesort pseudocode is:

```
if there are items remaining {
   mergesort the left half of the items
   mergesort the right half of the items
   merge the two halves into a completely sorted list
}
```

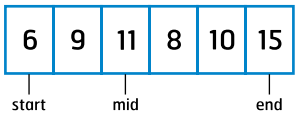
The mergesort subtasks are recursive calls to mergeSort() and a call to merge(). The mergesort() method will need arguments indicating which portion of the array is to be sorted. Similarly, merge() implements the merging of two halves and needs arguments indicating which portion of the array is to be merged.

The mergesort() method is defined as:

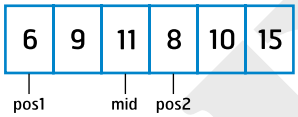
```
/**
 * Sorts items[start..end]
 * pre: start > 0, end > 0
 * post: items[start..end] is sorted low to high
 */
public static void mergesort(int[] items, int start, int end) {
  if (start < end) {
    int mid = (start + end) / 2;
    mergesort(items, start, mid);
    mergesort(items, mid + 1, end);
    merge(items, start, mid, end);
  }
}
```

The stopping condition for the recursive function is determined by comparing `start` and `end`. The middle of the array is calculated using integer division, which automatically truncates the decimal portion of the quotient.

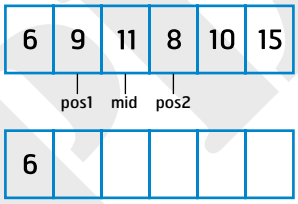
The merge() method uses a temporary array to store items moved from two sorted portions of the items array. The elements are moved so that the temporary array is sorted. To illustrate the merge() algorithm, suppose at entry to merge() the array looks like:



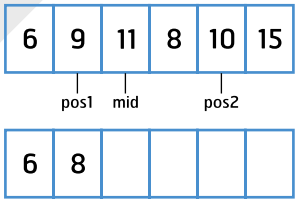
The array is sorted from `start` to `mid` and from `mid+1` to `end`. The merge() method starts by examining the first element of each sorted portion, `start` and `mid+1`, as indicated by `pos1` and `pos2`:



Since items[pos1] < items[pos2], the element items[pos1] is moved to the new array, and pos1 is incremented:



In this case, items[pos1] > items[pos2], so the the element items[pos2] is moved to the new array and pos2 incremented:



This process is repeated until all items have been moved. Since it is likely that one array portion will be exhausted before the other, merge() tests for this case and just moves items from the remaining list. Finally, merge() copies the merged items in the temporary array to the original array.

The Sorts class has been modified to include a mergesort() method. Note that the merge() method is `private` because it is a helper method:

```
/**
 * Merges two sorted portion of items array
 * pre: items[start..mid] is sorted. items[mid+1..end] sorted.
 * start <= mid <= end
 * post: items[start..end] is sorted.
 */
private static void merge(int[] items, int start,
                          int mid, int end) {
  int[] temp = new int[items.length];
  int pos1 = start;
  int pos2 = mid + 1;
  int spot = start;

  while (!(pos1 > mid && pos2 > end)) {
    if ((pos1 > mid) ||
        ((pos2 <= end) && (items[pos2] < items[pos1]))) {
      temp[spot] = items[pos2];
      pos2 += 1;
    } else {
      temp[spot] = items[pos1];
      pos1  += 1;
    }
    spot += 1;
  }
  /* copy values from temp back to items */
  for (int i =  start; i <= end; i++) {
    items[i] = temp[i];
  }

}


/**
 * Sorts items[start..end]
 * pre: start > 0, end > 0
 * post: items[start..end] is sorted low to high
 */
public static void mergesort(int[] items, int start, int end) {
  if (start < end) {
    int mid = (start + end) / 2;
    mergesort(items, start, mid);
    mergesort(items, mid + 1, end);
    merge(items, start, mid, end);
  }
}
```

The `if` in the merge() method says that if the pos1 (left) subarray is exhausted, or if the pos2 (right) subarray is not exhausted and the pos2 element is less than the pos1 element, then move an item from the pos2 subarray to the temp array, otherwise move an item from the pos1 subarray. This process continues until both subarrays are exhausted.

The TestSorts application has been modified to use the mergesort() method to sort an array of integers:

```
import java.util.Scanner;
import java.util.Random;

public class TestSorts {

  public static void displayArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + "  ");
    }
    System.out.println("\n");
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int numItems;
    int[] test;
    Random rand = new Random();

    System.out.print("Enter number of elements: ");
    numItems = input.nextInt();

    /* populate array with random integers */
    test = new int[numItems];
    for (int i = 0; i < test.length; i++) {
      test[i] = rand.nextInt(100);
    }
    System.out.println("Unsorted:");
    displayArray(test);

    Sorts.mergesort(test, 0, test.length - 1);

    System.out.println("Sorted:");
    displayArray(test);
  }
}
```

The TestSorts application produces output similar to:

```
Enter number of elements: 10
Unsorted:
99  55  94  59  75  26  71  16  91  3

Sorted:
3  16  26  55  59  71  75  91  94  99
```

Review: ObjectsMergesort

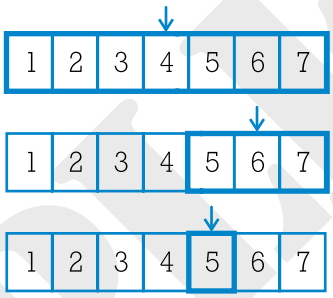
Create an ObjectsMergesort application that implements a mergesort on an array of objects. Test the sort on an array of String objects.

Binary Search

TIP The linear search algorithm was introduced in Chapter 10. A linear search, also called a sequential search, is much less efficient than a binary search. However, a linear search does not require a sorted list.

Arrays are sorted in order to perform a more efficient search. A binary search is used with a sorted list of items to quickly find the location of a value. Like the mergesort algorithm, the binary search algorithm also takes a divide-and-conquer approach. It works by examining the middle item of an array sorted from low to high, and determining if this is the item sought, or if the item sought is above or below this middle item. If the item sought is below the middle item, then a binary search is applied to the lower half of the array; if above the middle item, a binary search is applied to the upper half of the array, and so on.

For example, a binary search for the value 5 in a list of items 1, 2, 3, 4, 5, 6, and 7 could be visualized as:



The binary search algorithm is very efficient. For example, an array of 100 elements checks no more than 8 elements in a search, and in an array of one million items no more than 20 items are checked. If a list of the entire world's population were to be searched using this algorithm, less than 40 checks are made to find any one person.

The binary search algorithm can be implemented recursively. The pseudocode for a recursive solution appears like:

```
if (goal == items[mid]) {
   return(mid)
} else if (goal < items[mid]) {
   return(binarySearch(lowerhalf)
} else {
   return(binarySearch(upperhalf)
}
```

The Searches class on the next page implements a binary search:

```
public class Searches{

  /**
   * Searches items array for goal
   * pre: items is sorted from low to high
   * post: Position of goal has been returned,
   * or -1 has been returned if goal not found.
   */
  public static int binarySearch(int[] items, int start,
                                 int end, int goal) {

    if (start > end) {
      return(-1);
    } else {
      int mid = (start + end) / 2;
      if (goal == items[mid]) {
        return(mid);
      } else if (goal < items[mid]) {
        return(binarySearch(items, start, mid-1, goal));
      } else {
        return(binarySearch(items, mid+1, end, goal));
      }
    }
  }
}
```

The TestSorts application has been modified to sort an array of integers and then prompt the user for a number to search for:

```
import java.util.Scanner;
import java.util.Random;

public class TestSorts {

 public static void displayArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
     System.out.print(array[i] + "  ");
   }
   System.out.println("\n");
 }

 public static void sortIntArray() {
   Scanner input = new Scanner(System.in);
   int numItems, searchNum, location;
   int[] test;
   Random rand = new Random();

   System.out.print("Enter number of elements: ");
   numItems = input.nextInt();

   /* populate and sort array */
   test = new int[numItems];
   for (int i = 0; i < test.length; i++) {
     test[i] = rand.nextInt(100);
   }
   Sorts.mergesort(test, 0, test.length - 1);
   System.out.println("Sorted:");
   displayArray(test);
```

```
    /* search for number in sorted array */
    System.out.print("Enter a number to search for: ");
    searchNum = input.nextInt();
    while (searchNum != -1){
      location = Searches.binarySearch(test, 0,
                                       test.length-1, searchNum);
      System.out.println("Number at position: " + location);
      System.out.print("Enter a number to search for: ");
      searchNum = input.nextInt();
    }
  }


  public static void main(String[] args) {
    sortIntArray();
  }
}
```

The modified TestSorts application displays output similar to:

```
Enter number of elements: 10
Unsorted:
80  84  68  23  36  70  75  39  29  11

Sorted:
11  23  29  36  39  68  70  75  80  84

Enter a number to search for: 11
Number at position: 0
Enter a number to search for: 23
Number at position: 1
Enter a number to search for: 81
Number at position: -1
Enter a number to search for: 84
Number at position: 9
Enter a number to search for: 68
Number at position: 5
Enter a number to search for: -1
```

Review: SearchLocations

Create a SearchLocations application that displays the positions examined during a binary search. The application output should look similar to:

```
1  4  4  7  10  13  15  16  24  24  25  27  27  30  30  30
49  50  51  52  53  54  54  55  57  58  64  69  74  78  79
85  86  86  88  89  96  96  97  98

Enter a number to search for: 55
Examining 24
Examining 37
Examining 30
Examining 27
Examining 28
Number at position: 28
Enter a number to search for:
```

Review: ObjectsBinarySearch

Create an ObjectsBinarySearch application that implements a binary search on an array of objects. Test the search on an array of String objects.

Review: BinarySearch2

Create a BinarySearch2 application that implements a nonrecursive binary search. The binary search algorithm can be implemented without recursion by doing the following:

- Enclose the method body with a `do-while` loop
- Replace the recursive calls with appropriate assignments to the values of `start` or `end`.

Depth-First Searching

Many programs generate and search through a series of possibilities, such as:

- different paths through a maze
- different possible ways of making change
- different possible plays in a game
- different schedules for a student in a school
- different pixels reachable from an initial pixel

All these tasks can be solved through the recursive technique called *depth-first searching*. The depth-first searching algorithm works by searching from a given starting position, processing that position, and then (recursively) searching from all adjacent positions.

Depth-first searching can be illustrated in a DetectColonies application that allows a researcher to determine the number and size of distinct colonies of bacteria on a microscope slide. The slide has been converted to digital format, where a `*` represents a cell that is part of a colony, and a `-` represents the background color of the slide. A slide file has the format:

First line: length of slide
Second line: width of slide
Remaining lines: slide data.

For example, a slide file could look similar to:

```
7
9
---------
---*--*--
--***-**-
***---*--
**-**----
---****--
---*--*--
```

Cells are considered to be part of the same colony if they touch horizontally or vertically. The example slide contains three colonies. Counting rows and columns starting from zero, one colony has a cell at (1, 3) and contains 9 elements. Another colony has four elements with a cell at (1, 6), and the third has eight elements with a cell at (4, 3). Note that the first and third colonies are considered separate because they touch across a diagonal but not horizontally or vertically.

The DetectColonies application analyzes the slide and displays the following output:

```
Colony at (1,3) with size 9
Colony at (1,6) with size 4
Colony at (4,3) with size 8
```

A depth-first search is appropriate here because once a colony cell is detected, all the possible directions for connected cells must be searched. If a connected cell is a colony cell, then all the possible directions for that cell must be searched, and so on. The basic idea is that, given a starting cell at (row, col) in a colony, the total number of connected cells in that colony can be found as:

1 for the starting cell

+ count of connected cells starting with (row+1, col)

+ count of connected cells starting with (row–1, col)

+ count of connected cells starting with (row, col+1)

+ count of connected cells starting with (row, col–1)

The latter four lines are recursive calls. To find the starting cells, each cell in the slide is tested with a nested `for` loop.

When implementing a depth-first search algorithm, code must be included to avoid infinite recursion. For example, a starting cell of (1, 3) generates a recursive call with cell (2, 3), which generates a call with (1, 3) again, which generates a call with (2, 3), and so on. For this application, a cell will be changed to the background color once it has been examined. This makes counting colonies a destructive algorithm.

The DetectColonies application can be modeled with a Slide class. The constructor will load the slide data into variable member `slideData`, which is a two-dimensional array appropriate for modeling the slide. Method members displaySlide() and displayColonies() will display the slide and determine the colonies. Member constants `COLONY` and `NON_COLONY` will represent * and –.

The displayColonies() method checks each cell of the entire slide, and whenever a colony cell is encountered, displayColonies() determines the colony size and displays data about the colony. To determine the size, displayColonies() calls a private member method collectCells(), which changes a cell to the background once it is counted.

The Slide class is implemented on the next page:

```
import java.io.*;

public class Slide {
  private char COLONY = '*', NON_COLONY = '-';
  private char[][] slideData;

  /**
   * constructor
   * pre: Slide file contains valid slide data in the format:
   * first line: lenght of slide
   * second line: width of slide
   * remaining lines: slide data
   * post: Slide data has been loaded from slide file.
   */
  public Slide(String s) {

    try {
      File slideFile = new File(s);
      FileReader in = new FileReader(slideFile);
      BufferedReader readSlide = new BufferedReader(in);
      int length = Integer.parseInt(readSlide.readLine());
      int width = Integer.parseInt(readSlide.readLine());
      slideData = new char[length][width];
      for (int row = 0; row < length; row++) {
        for (int col = 0; col < width; col++) {
          slideData[row][col] = (char)readSlide.read();
        }
        readSlide.readLine(); //read past end-of-line
      }
      readSlide.close();
      in.close();
    } catch (FileNotFoundException e) {
      System.out.println("File does not exist or could not
                          be found.");
      System.err.println("FileNotFoundException: "
                          + e.getMessage());
    } catch (IOException e) {
      System.out.println("Problem reading file.");
      System.err.println("IOException: " + e.getMessage());
    }
  }


  /**
   * Determines a colony size
   * pre: none
   * post: All colony cells adjoining and including
   * cell (Row, Col) have been changed to NON_COLONY,
   * and count of these cells is returned.
   */
  private int collectCells(int row,  int col) {

    if ((row < 0) || (row >= slideData.length) ||
        (col < 0) || (col >= slideData[0].length)
        || (slideData[row][col] != COLONY)) {
      return(0);
    } else {
        slideData[row][col] = NON_COLONY;
        return(1+
          collectCells(row+1, col)+
          collectCells(row-1, col)+
          collectCells(row, col+1)+
          collectCells(row, col-1));
    }
  }
```

```
  /**
   * Analyzes a slide for colonies and displays colony data
   * pre: none
   * post: Colony data has been displayed.
   */
  public void displayColonies() {
    char[][] temp;
    int count;

    for (int row = 0; row < slideData.length - 1; row++) {
        for (int col = 0; col < slideData[0].length; col++) {
          if (slideData[row][col] == COLONY) {
            count = collectCells(row, col);
            System.out.println("Colony at (" + row + "," + col
                                + ") with size " + count);
          }
        }
    }
  }


  /**
   * Displays a slide.
   * pre: none
   * post: Slide data has been displayed.
   */
  public void displaySlide() {

    for (int row = 0; row < slideData.length; row++) {
      for (int col = 0; col < slideData[0].length; col++) {
        System.out.print(slideData[row][col]);
      }
      System.out.println();
    }
  }
}
```

The depth-first search algorithm is implemented in the helper method collectCells(). The first `if` statement checks to see that the current position is on the slide and contains a colony cell. This eliminates the need to check before each recursive call. As good programming style, it is better to check data at the start of the recursive function rather than before each call.

The DetectColonies application is relatively simple:

```
public class DetectColonies {

  public static void main(String[] args) {

    Slide culture = new Slide("slide.dat");
    culture.displaySlide();
    culture.displayColonies();
  }
}
```

The DetectColonies application displays output similar to:

```
---------
---*--*--
--***-**-
***---*--
**-**----
---****--
---*--*--
Colony at (1,3) with size 9
Colony at (1,6) with size 4
Colony at (4,3) with size 8
```

Review: DetectColonies – part 1 of 3

How must the DetectColonies application be modified if the definition of a colony allowed the colony to be connected across diagonals? What colonies would be reported by DetectColonies for the sample slide?

Review: DetectColonies – part 2 of 3

Modify the DetectColonies application to display the slide colonies from largest to smallest.

Review: DetectColonies – part 3 of 3

What slide will be output by the second `culture.displaySlide()` statement if DetectColonies contained the statements below? Explain.

```
public static void main(String[] args) {

    Slide culture = new Slide("slide.dat");
    culture.displaySlide();
    culture.displayColonies();
    culture.displaySlide();
}
```

Algorithm Analysis

TIP Theoretical running times can be written in Big Oh notation, which is a theoretical measure of an algorithm's efficiency.

Algorithm analysis includes measuring how efficiently an algorithm performs its task. A more efficient algorithm has a shorter running time. *Running time* is related to the number of statements executed to implement an algorithm. It can be estimated by calculating statement executions. Running time estimations are usually based on a worst-case set of data. For example, an array that is already sorted or nearly sorted may require fewer statement executions than an array of data that is in reverse sorted order. Since the original state of the data is usually unknown, the worst case should be assumed.

As a first analysis, consider the selection sort algorithm, which uses nested `for` loops to sort items. If the sort is thought of in simplified terms, with each `for` statement containing a single statement, for an array of n items, $n * n$ statements are executed. The selection sort is therefore said to have a running time of n^2.

The insertion sort algorithm seems more efficient because a `while` loop is used within a `for` loop. This could allow for a faster sort in some cases, but in the worst case, the insertion sort algorithm executes n statements * n–1 statements for a running time of about n^2.

The mergesort is a more complicated algorithm, but a divide and conquer approach can be much more efficient than a linear approach. Because this algorithm divides the array and each subarray in half until the base case of one element is reached, there are $Log_2 n$ calls to mergesort() and then n calls to merge. The mergesort algorithm is said to have running time of n $Log_2 n$.

The binary search also implements a divide and conquer approach. However, the elements are already ordered. The algorithm is simply performing a search. Because this algorithm divides the array and each subarray in half until the base case of one element is reached, there are $Log_2 n$ calls to binarySearch(). Therefore, the algorithm is said to have a running time of $Log_2 n$.

Chapter Summary

Sorting is the process of putting items in a designated order, either from low to high or high to low. The selection sort, insertion sort, and mergesort algorithms were presented in this chapter. The merge sort algorithm takes a divide-and-conquer approach to sorting. It is implemented recursively and is much faster than the selection and insertion sorts.

Objects cannot be compared with relational operators. Therefore, lists of objects are sorted by implementing an algorithm that uses the Comparable interface. Interfaces can be used as data types, which allows a generic implementation of an algorithm that sorts objects. Objects that are to be sorted must have a class that implements the Comparable interface.

A method can call itself in a process called recursion. Recursion is a programming technique that can be used whenever a problem can be solved by solving one or more smaller versions of the same problem and combining the results. To prevent infinite recursion, a base case that requires no recursion must be part of the recursive solution.

A binary search algorithm can be used to find an element in a sorted array. Binary search is implemented recursively and is very efficient. The depth-first searching algorithm can be used to search through a series of possibilities. When implementing depth-first searching, code must be included to avoid infinite recursion.

Algorithm analysis includes how efficiently an algorithm performs its task. A more efficient algorithm has a shorter running time.

Vocabulary

Binary search A searching algorithm that recursively checks the middle of a sorted list for the item being searched for.

Base case The part of a recursive solution that requires no recursion.

Depth-first search A searching algorithm that recursively checks a starting position, processes that position, and then searches adjacent positions.

Infinite recursion A recursive solution which has no base case.

Mergesort A sorting algorithm that recursively divides a list into halves, sorting those halves, and then merging the lists so that the items are in order.

Insertion sort A sorting algorithm that repeatedly inserts an item into the appropriate position of a subarray until the subarray has no items left to insert.

Recursion The process in which a method calls itself. A programming technique that can be used whenever a problem can be solved by solving one or more smaller versions of the same problem and combining the results.

Recursive call A call to a method from within the same method.

Selection sort A sorting algorithm that repeatedly selects the lowest item in a subarray of an array and moves it to the position just before the subarray until the subarray has no items left to search.

Java

Comparable A java.lang interface that is required to be implemented by a class if their objects are to be sorted. Comparable can also be used as a data type.

Critical Thinking

1. For the list 4, 6, 2, 10, 9, show how the numbers are ordered after each loop iteration of the algorithms:
   a) selection sort
   b) insertion sort

2. What must be done to a list of items in order to use a binary search to find a specific item?

3. Use the recursive method below to answer the questions:

```
public void ct(int n) {
  System.out.println("Starting " + n);
  if (n > 0) {
    ct(n/3);
    System.out.println("Middle " + n);
  }
}
```

   a) What output is generated when ct(13) is called?
   b) What output is generated when ct(3) is called?
   c) What output is generated when ct(0) is called?

4. Use the recursive method below to answer the questions:

```
public void ct(int n) {
  System.out.println(n);
  if (n > 0) {
    if (n % 2 == 1) {
      ct(n/3);
    } else {
      ct(n/2);
    }
  }
}
```

   a) What output is generated when ct (13) is called?
   b) What output is generated when ct(14) is called?
   c) What output is generated when ct(15) is called?

5. Use the recursive method below to answer the questions:

```
public void ct(int n) {
  if (n > 0) {
    ct(n/10);
    System.out.println(n % 10);
  }
}
```

   a) What output is generated when ct (13) is called?
   b) What output is generated when ct(124) is called?
   c) What output is generated when ct(21785) is called?
   d) What in general does this method do?

6. Use the recursive method below to answer the questions:

```
public void whatzItDo() {
  Scanner input = new Scanner(System.in);
  String letter = input.next();
  if (!letter.equals(".")) {
    whatzItDo();
    System.out.print(letter);
  }
}
```

   a) What output is generated when the user enters `T`, `E`, `S`, `T`, `.` ?
   b) What in general does this method do?

7. A sorting algorithm is said to be "stable" if two items in the original array that have the same "key value" (the value to be sorted on) maintain their relative position in the sorted version. For example, assume an array with the following data:

Ann	Jon	Mel	Tom	Kim
20	19	18	19	22

   When the array is sorted by age, a stable sort would guarantee that Jon would stay ahead of Tom in the sorted array, as in:

Kim	Ann	Jon	Tom	Mel
22	20	19	19	18

   and not:

Kim	Ann	Tom	Jon	Mel
22	20	19	19	18

   Which of the sorts presented in this chapter (selection, mergesort) is stable? For each which is not stable, give an example of data to illustrate this.

True/False

8. Determine if each of the following are true or false. If false, explain why.
   a) Sorting always puts items in order from low to high.
   b) One measure of the efficiency of a sorting algorithm is the speed at which it can complete a sort.
   c) The selection sort algorithm is more efficient than the insertion sort.
   d) A method can call itself.
   e) The merge sort algorithm is more efficient than the selection sort algorithm for large arrays.
   f) A binary search is used to sort a list of items.
   g) A binary search starts by examining the last item in an array.
   h) The more statements required to complete a task, the more efficient the algorithm.
   i) An interface can be used as a data type.
   j) A recursive solution that has no base case results in infinite recursion.
   k) An insertion sort recursively divides a list into halves, sorting those halves, and then merging the lists in order.

Exercises

Exercise 1 — Friends

Create a Friends database application that maintains a file of Friend objects that contain names, telephone numbers, and email addresses. The Friends application should load Friend records from a file and then allow the user to add new friends, delete friends, display a list of all friends by either first name or last name, and search for a friend. The application should display a menu similar to:

```
1. Add a friend.
2. Display friends by last name.
3. Display friends by first name.
4. Find a friend.
5. Delete a friend.
6. Quit.
```

Exercise 2 — TernarySearch

Modify the Searches class to include a ternarySearch() method. A ternary search, similar to a binary search, divides an array into three pieces rather than two. A ternary search finds the points that divide the array into three roughly equal pieces, and then uses these points to determine where the goal should be searched for.

Exercise 3 — InterpolationSearch

Modify the Searches class to include an interpolationSearch() method. An interpolation search is a variation of the binary search. The idea is to look in a likely spot, not necessarily the middle of the array. For example, if the value sought is 967 in an array that holds items ranging from 3 to 1022, it would be intelligent to look nearly at the end of the array. Mathematically, because 967 is about 95% of the way from 3 to 1022, the position to start searching at is a position 95% of the way down the array. For example, if the array holds 500 elements, the first position to examined is 475 (95% of the way from 1 to 500). The search then proceeds to a portion of the array (either 1..474 or 476..500) depending upon whether 967 is greater or less than the 475th element.

Exercise 4 — NumDigits

Create a NumDigits application that includes a recursive method numDigits() that returns the number of digits in its integer parameter. Numbers –9 through 9 have one digit; numbers –99 to –10 and 10 to 99 have two digits, and so on. (Hint: the number of digits of a number `n` is one more than the number of digits in `n/10`.)

Exercise 5 — DetectColonies2

Create a DetectColonies2 application that is based on DetectColonies presented in this chapter. This application gives improved colony results because slides are now digitized to report color. For example, a slide file could look similar to:

```
6
8
00550000
00050000
00005500
01200000
01111000
00000030
```

The digits 1 through 9 represent various colors. The digit 0 represents black (background color).

The DetectColonies2 application should display a listing of the size, location, and color value of each colony on the slide. A colony is defined as a connected (horizontally or vertically) sequence of cells holding the same color value. For the above slide, the application should report:

Color	Size	Location
5	3	1, 3
5	2	3, 5
1	5	4, 2
2	1	4, 3
3	1	6, 7

Exercise 6 — Knapsack

A well-known problem in computer science is called the *knapsack problem*. A variation is as follows:

> Given a collection of weights of (possibly) different integral values, is it possible to place some of the weights in a knapsack so as to fill it to some exact total weight?
>
> For example, if the weights are 3, 5, 6, and 9, then it is possible for such totals as 3, 8, 11, 14, 17, etc. to be made exactly, but 2, 4, 22, etc. are not possible.

Create a Knapsack application that solves this problem. The fillKnapsack() method handles the first weight, and then recursively handles the remaining weights with an isPossible() helper method. fillKnapsack() and isPossible() have the following declarations:

```
/* Returns true if there exists a subset of the items in
 * weights[start..weights.length] that sum to goal.
 * pre: items in weights[start..weights.length] > 0
 * post: true has been returned if there exists a subset
 * of items in weights[start..weights.length] that sum to goal.
 */
fillKnapsack(int[] weights, int goal, int start)
```

```
/* Returns true if there exists a subset of the items in
 * weights that sum to goal.
 * pre: items in weights > 0
 * post: true has been returned if there exists a subset
 * of items in weight that sum to goal.
 */
isPossible(int[] weights, int goal)
```

The fillKnapsack algorithm determines if the goal can be found in all of the items not including the first, or if it can be found by including the first in the subset. The pseudocode is:

```
if (simple case) {
    handle simple cases
} else {
    if (fillKnapsack(weights, goal, start+1)) {
        return(true);
    } else if (fillKnapsack(weights, goal-weights[start], start+1)) {
        return(true);
    } else {
        return(false);
    }
}
```

Note that the simple cases will need to be determined and handled properly.

Exercise 7 — Maze

A maze can be defined in a file using `X` characters to represent walls, space characters to represent paths, and a `$` character to represent the goal. For example, a file containing a maze could look similar to:

```
8
10
XXXXXXXXXX
X        X
XX XXX XXX
XX   X   X
XXXX X X X
X    X XXX
X XXXX  $X
XXXXXXXXXX
```

The starting point is assumed to be location (1, 1) and the maze is assumed to have a border. Create a Maze application that displays the sequence of positions in a path from the start to the goal, or indicate if no path is available. For example, for the maze above, the application should report:

```
Path:  (1,1)  (1,2)  (1,3)  (1,4)  (1,5)  (1,6)
(2,6)  (3,6)  (4,6)  (5,6)  (6,6)  (6,7)  (6,8)
```

Exercise 8 — MyBoggle

The game of Boggle is played on a square board with random letters. The object is to find words formed on the board by contiguous sequences of letters. Letters are considered to be touching if they are horizontally, vertically, or diagonally adjacent. For example, the board:

```
Q W E R T
A S D F G
Z X C V B
Y U A O P
G H J K L
```

contains the words WAS, WAXY, JOB, and others, but not the word BOX. Words can contain duplicate letters, but a single letter on the board may not appear twice in a single word, for example POP is not contained on this board.

Create a MyBoggle application that displays a board of random letters, and allows the user to enter words found on the board. The application should report if the word entered by the user is indeed on the board.

Hint: Search the board for the first letter of the word entered, and then recursively search around the found letter for the remaining letters of the word.