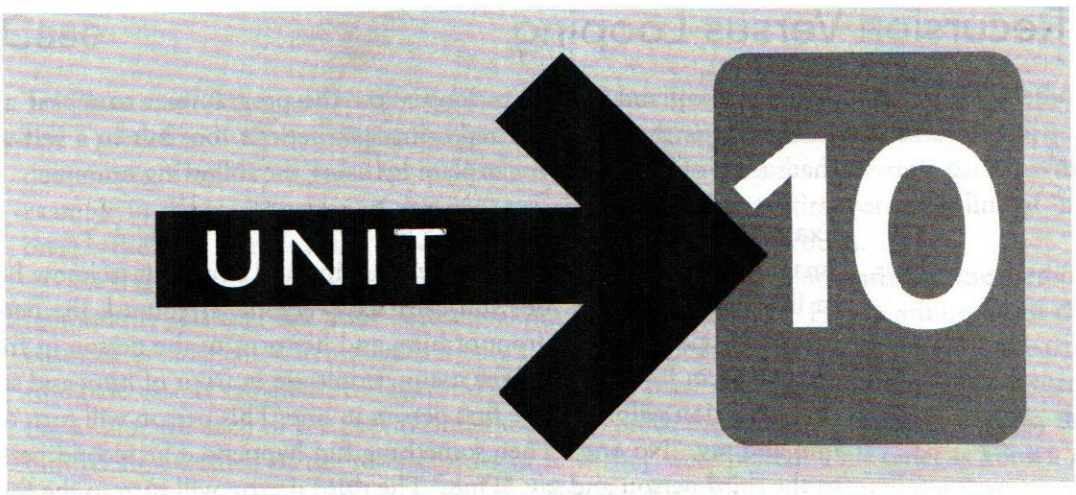


Recursion

IN THIS UNIT

Summary: This unit defines and explains recursion. It also shows how to read and trace a recursive method. You will also learn how to implement recursive algorithms for searching and sorting.

Key Ideas

- A recursive method is a method that calls itself.
- Recursion is not the same as nested `for` loops or `while` loops.
- A recursive method must have a base case that tells the method to stop calling itself and a recursive call.
- The Binary Search is an efficient way to search for a target in a sorted list.
- The Merge Sort is a recursive sorting routine.
- You will need to know how to trace recursive methods on the AP Computer Science A Exam. You will not have to write recursive methods.

Recursion Versus Looping

The `for` loop and the `while` loop allow the programmer to repeat a set of instructions. **Recursion** is the process of repeating instructions too, but in a **self-similar** way. I know that's kind of weird. Let me explain by using the following example.

Example 1

Imagine you are standing in a really long line and you want to know how many people are in line in front of you, but you can't see everyone. If you ask the person in front of you how many people are in front of him, and he turns to the person in front of him and asks the person in front of him how many people are in front of him, and so on, eventually this chain reaction will reach the first person in line. This person will turn to the person behind him and say, "No one." Then something fun happens. The second person in line will turn to the third person and say, "One." The third person will turn to the fourth person and say, "Two," and so on, until finally the person in front of you will respond with the number of people who are in front of him. This creative solution to the problem is an example of how a recursive algorithm works.

A **recursive** method is a method that calls itself. Recursion is a programming technique that can be used to solve a problem in which a solution to the problem can be found by making subsequently smaller iterations of the same problem. When used correctly, it can be an extremely elegant solution to a problem. And by the way, a recursive solution can be written by using iteration, but we still need to know it because it does make some problems much easier to write and understand.

We know that one method can call another method, right? What if the method called itself? If you stop and think about that, after the method was **invoked** the first time, the program would never end because the method would continue to call itself forever (think of the movie *Inception*).

Example 2

Here's an example of a really bad recursive method.

```
public static void IAmABadRecursiveMethod()
{
   IAmABadRecursiveMethod();    // This method never stops calling itself
}
```

However, if the method had some kind of trigger that told it when to stop calling itself, then it would be a good recursive method.

General Form for a Recursive Method

```
public static returnType goodRecursiveMethod(parameterList)
{
   if ( /* base case is true */)
      return something;
   else
      return goodRecursiveMethod(argumentList);
}
```

Note: All recursive methods must have a base case and a recursive call.

The Base Case

The **base case** of a recursive method is a comparison between a parameter of the method and a predefined value strategically chosen by the programmer. The base case comparison determines whether or not the method should call itself again. Referring back to the example in the introduction of this concept, the response by the first person in line of "no one" is the base case. It is the answer that reverses the recursive process.

Each time the recursive method calls itself and the base case is not satisfied, the method temporarily sets that call aside before making a new call to itself. This continues until the base case is satisfied. When there is finally a call to the method that makes the base case true (the base case is satisfied), the method stops calling itself and starts the process of returning a value for each of the previous method calls. Since the calls are placed on a **stack**, the returns are executed in the reverse order of how they were placed. This order is known as last-in, first-out (LIFO), which means that the last recursive call made is the first one to return a value.

Furthermore, each successive call of the method should bring the value of the parameter *closer and closer to making the base case true.* This means that with each call to the method, the value of the parameter should be *closing in* on making the base case true rather than moving farther away from it being true.

If a recursive method does not contain a valid base case comparison, then it may continue to call itself into oblivion. That would be bad.

What do I mean by bad? Remember that every time a recursive method is called, it has to set that call aside temporarily until the base case is satisfied. This process is called *putting the call on the stack* and the computer stores this stack in its RAM. If the base case is *never* satisfied, then the method calls pile up and the computer eventually runs out of memory. This will cause a **stack overflow error** and it occurs when you have an **infinite recursion.**

The Base Case

The base case is a comparison between a parameter of the method and some specific predefined value chosen by the programmer. When the base case is true, the method stops calling itself and starts the process of returning values. Every recursive method needs a valid base case or else it will continue to call itself indefinitely (infinite recursion) and cause an out-of-memory error, called a stack overflow error.

Example: The Factorial Recursive Method

A very popular problem to solve using recursion is the factorial calculation. The factorial of a number uses the notation **n!** and is calculated by multiplying the integer n by each of the integers that precede it all the way down to 1. Factorial is useful for computing combinations and permutations when doing probability and statistics.

Example

Find the answer to 4! and 10!

4! = 4 * 3 * 2 * 1 = 24. Therefore 4! is 24.
10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 3628800. Therefore, 10! is 3628800.

Why the Factorial Calculation Is a Good Recursive Example

The definition of a factorial can be written recursively. This means that the factorial of the number, n, can use the factorial of the previous number, n − 1, in its computation.

$$n! = n * (n - 1)! \qquad \text{where } 1! \text{ is defined as } 1$$

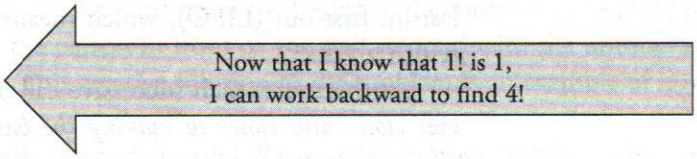
Example

Write 4! recursively using math.

When you start to compute 4! using the definition of factorial, you see that 4! is 4 times 3! This means that in order to compute 4!, you need to figure out what 3! is. But 3! is 3 times 2! so we have to figure out what 2! is. And 2! is 2 times 1!, and 1! is 1.

Do you see how we had to put our calculations on hold until we got to 1!? Once we got a final answer of 1 for 1!, we were able to compute 4! by working backward.

1! = 1
so 2! = 2 * 1! = 2 * 1 = 2
so 3! = 3 * 2! = 3 * 2 = 6
so 4! = 4 * 3! = 4 * 6 = 24



Implementation of the Factorial Recursive Method

```
public class Factorial
{
   public static void main(String[] args)
   {
      System.out.println(factorial(5));     // find factorial(5)
   }

   /**
    * This is a recursive method that computes the
    * factorial of a number.
    *
    * @param n the number we are finding the factorial of
    * @return the answer to the factorial of n
    *
    * PRECONDITION:  n >= 1
    * POSTCONDITION:  The result of the recursive call
    *                 is the factorial of the original argument
    */
   public static int factorial(int n)
   {
      if (n == 1)                           // base case comparison
         return 1;
      else
         return n * factorial(n - 1);       // move closer to base case
   }
}
```

```
OUTPUT
120
```

Good News, Bad News
On the AP Computer Science A exam, you will never be asked to write your own original recursive method on the Free-Response section. However, on the Multiple-choice section you will be asked to hand-trace recursive methods and state the results.

Hand-Tracing the Factorial Recursive Method

This is kind of tricky, so read it over very carefully. When tracing a recursive method call, write out each call to the method including its parameter. When the base case is reached, replace the result from each method call with its result, one at a time. This process will force you to calculate the return values in the reverse order from the way that the calls were made.

Goal: Compute the value of factorial(5) using the process of recursion.

Step 1: This problem is similar to the "find the number of people in line" example. The answer to a factorial requires knowledge of the factorial preceding it.

Suppose I ask the number 5, "Hey what's your factorial?" 5 responds with, "I don't know. I need to ask 4 what its factorial is before I can find out my own." Then 4 asks 3 what its factorial is. Then 3 asks 2 what its factorial is. Then 2 asks 1 what its factorial is. Then 1 says, "1".

Step 2: After 1 says, "1", the process reverses itself and each number answers the question that was asked of it. The answer of "1" is the base case.

1 says, "My factorial is 1."

2 then says, "My factorial is 2 because 2 * 1 is 2."

3 then says, "My factorial is 6 because 3 * 2 is 6."

4 then says, "My factorial is 24 because 4 * 6 is 24."

And finally, 5 turns to me and says, "My factorial is 120, because 5 * 24 is 120."

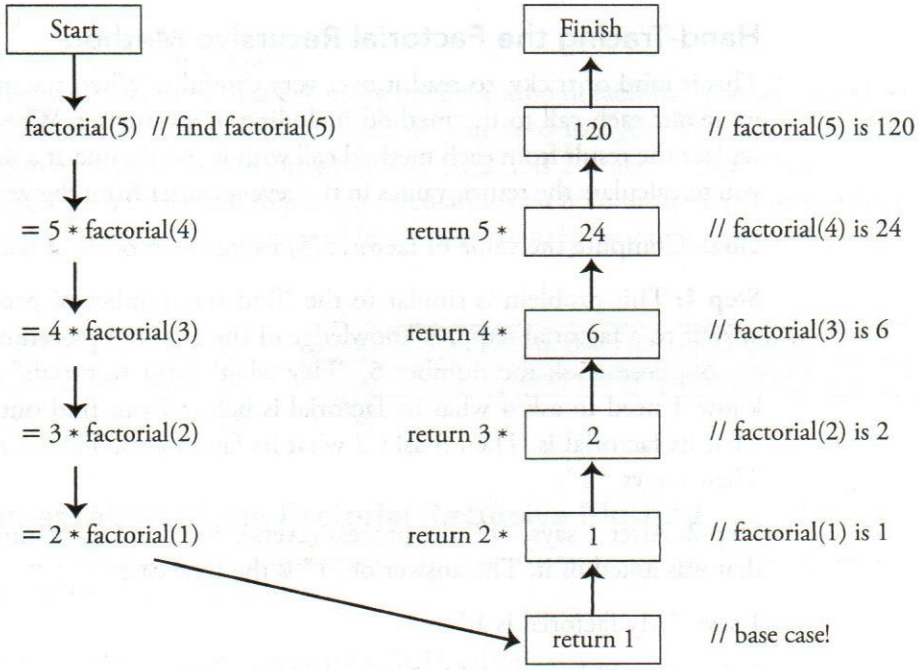
factorial(1) = 1
factorial(2) = 2 * factorial(1) = 2 * 1 = 2
factorial(3) = 3 * factorial(2) = 3 * 2 = 6
factorial(4) = 4 * factorial(3) = 4 * 6 = 24
factorial(5) = 5 * factorial(4) = 5 * 24 = 120

Conclusion: factorial(5) = 120

A Visual Representation of Computing a Recursive Value

Example

Explain how to compute the value of factorial(5) for the visual learner. As you trace through this problem, think of the original example of the people in the long line.



Each recursive call has its own set of local variables, including the formal parameters. In the example above a new formal parameter n gets created with each call to the factorial method. The parameter values capture the progress of a recursive process, much like loop control variables capture the progress of the loop. Once the parameter n reaches the value 1, the recursion reaches the base case and ends.

Example: The Fibonacci Recursive Method

On the AP Computer Science A exam, you will need to be able to hand-trace recursive method calls that have either multiple parameters or make multiple calls within the return statement. The following example shows a recursive method that has multiple calls to the recursive method within its return statement.

The Fibonacci Sequence

This popular sequence is often covered in math class as it has some interesting applications. In the Fibonacci sequence, the first two terms are 1, and then each term starting with the third term is equal to the sum of the two previous terms.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55 . . .

The Java code that is provided below is a recursive method that returns the value of the nth term in a Fibonacci sequence. **Example:** fibonacci(6) is 8. The value of the sixth term is 8.

```
public class AdvancedRecursiveProblem
{
   public static void main(String[] args)
   {
      System.out.println(fibonacci(6));              // find fibonacci(6)
   }

   /**
    * This is a recursive method that has multiple base cases and
    * uses two recursive calls within the return statement.
    *
    * @param n The fibonacci term to be calculated
    * @return The nth term of the fibonacci sequence
   */
   public static int fibonacci(int n)
   {
      if (n == 1 || n == 2)
         return 1;
      else
         return fibonacci(n - 1) + fibonacci(n - 2);
   }
}
```

```
OUTPUT
8
```

Hand-Tracing the Fibonacci Recursive Method

This is harder to follow than the factorial recursive method because the return statement includes two calls to the recursive method. Also, there are two base cases.

Goal: Compute the value of fibonacci(6).

Step 1: In a manner similar to the previous example, I ask 6, "What's your fibonacci?" 6 responds by saying, "I don't know right now, I need to ask both 5 and 4 what their fibonaccis are." In response, 5 says, "I need to ask both 4 and 3 what their fibonaccis are." And, 4 says, "I need to ask both 3 and 2 what their fibonaccis are." This continues until 2 and 1 are asked what their fibonaccis are.

Step 2: Ultimately, 2 and 1 will respond with "1" and "1" and each number can then answer the question that was asked of them. The 2 and 1 are the base cases for this problem.

fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = fibonacci(2) + fibonacci(1) = 1 + 1 = 2
fibonacci(4) = fibonacci(3) + fibonacci(2) = 2 + 1 = 3
fibonacci(5) = fibonacci(4) + fibonacci(3) = 3 + 2 = 5
fibonacci(6) = fibonacci(5) + fibonacci(4) = 5 + 3 = 8

Conclusion: fibonacci(6) is 8

Merge Sort

The **Merge Sort** is called a "divide and conquer" algorithm and uses recursion. The Merge Sort algorithm repeatedly divides the list into two groups until it can't do it anymore (that's where the recursion comes in). Next, the algorithm merges the smaller groups together and sorts them as it joins them. This process repeats until all the groups form one sorted group.

Merge Sort is a relatively fast sort and is much more efficient on large data sets than Insertion Sort or Selection Sort. Although it seems like Merge Sort is the best of these three sorts, its downside is that it requires more memory to execute.

Example

Sort a group of numbers using the Merge Sort algorithm.

Goal: Sort these numbers from smallest to largest: 8, 5, 2, 6, 9, 1, 3, 4

Step 1: Split the group of numbers into two groups: 8, 5, 2, 6 and 9, 1, 3, 4
Step 2: Split each of these groups into two groups: 8, 5 and 2, 6 and 9, 1 and 3, 4
Step 3: Repeat until you can't anymore: 8 and 5 and 2 and 6 and 9 and 1 and 3 and 4
Step 4: Combine two groups, sorting as you group them: 5, 8 and 2, 6 and 1, 9 and 3, 4
Step 5: Combine two groups, sorting as you group them: 2, 5, 6, 8 and 1, 3, 4, 9
Step 6: Repeat the previous step until you have one sorted group: 1, 2, 3, 4, 5, 6, 8, 9

Implementation

The following class contains three interdependent methods that sort a list of integers from smallest to greatest using the Merge Sort algorithm. The method `setUpMerge` is a recursive method.

```
public class MergeSort
{
   private static int[] myArray;
   private static int[] tempArray;

   public static void main(String[] args)
   {
      int[] inputArray = {8, 5, 2, 6, 9, 1, 3, 4};
      mergeSort(inputArray);
      for (int i: inputArray)
      {
         System.out.print(i + "\t");
      }
   }

   /**
    * This method starts the process of the Merge Sort.
    *
    * @param arr  the array to be sorted
    */
   private static void mergeSort(int arr[])
   {
      myArray = arr;
      int length = arr.length;
```

```
        tempArray = new int[length];
        setUpMerge(0, length - 1);
    }

    /**
     * This method is called recursively to divide the array
     * into each of the groups to be sorted.
     *
     * @param lower  the lower index of the subgroup
     * @param higher  the higher index of the subgroup
     */
    private static void setUpMerge(int lower, int higher)
    {
        if (lower < higher)
        {
            int middle = lower + (higher - lower) / 2;
            setUpMerge(lower, middle);
            setUpMerge(middle + 1, higher);
            doTheMerge(lower, middle, higher);
        }
    }

    /**
     * This method merges the elements in two subgroups
     * ([start, middle] and [middle + 1, end]) in ascending order.
     * @param lower the index of the lower element
     * @param middle the index of the  middle element
     * @param higher the index of the higher element
     */
    private static void doTheMerge(int lower, int middle, int higher)
    {
        for (int i = lower; i <= higher; i++)
        {
            tempArray[i] = myArray[i];
        }
        int i = lower;
        int j = middle + 1;
        int k = lower;
        while (i <= middle && j <= higher)
        {
            if (tempArray[i] <= tempArray[j])
            {
                myArray[k] = tempArray[i];
                i++;
            }
            else
            {
                myArray[k] = tempArray[j];
                j++;
            }
            k++;
        }
```

```
        while (i <= middle)
        {
            myArray[k] = tempArray[i];
            k++;
            i++;
        }
    }
}
```

```
OUTPUT
1   2   3   4   5   6   8   9
```

Binary Search

So far, the only algorithm we know to search for a target in a list is the Sequential Search. It starts at one end of a list and works toward the other end, comparing each element to the target. Although it works, it's not very efficient. The **Binary Search** algorithm is the most efficient way to search for a target item *provided the list is already sorted.* The binary search algorithm starts at the middle of a sorted array or `ArrayList` and eliminates half of the array or `ArrayList` in each iteration (either the "low" side of the array or the "high" side of the array) until the desired value is found or all elements have been eliminated.

Sorted Data
The Binary Search only works on sorted data. Performing a Binary Search on unsorted data produces invalid results.

Example
The "Guess My Number" game consists of someone saying something like, "I'm thinking of a number between 1 and 100. Try to guess it. I will tell you if your guess is too high or too low."

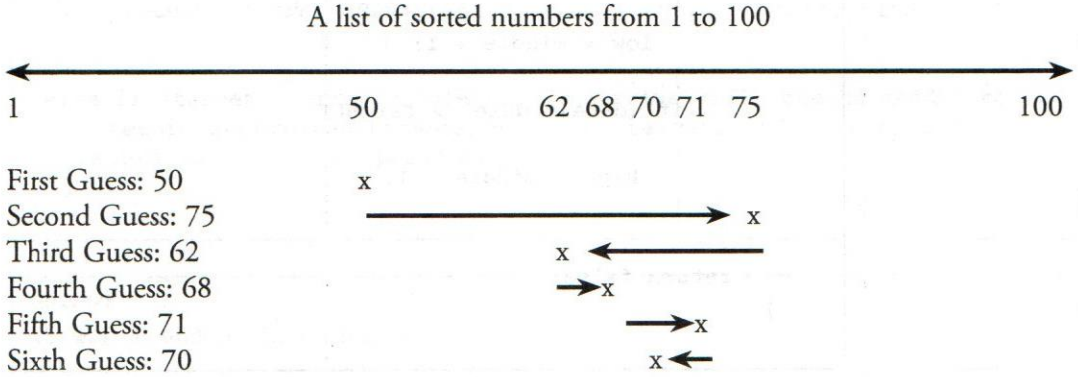
Solution 1: Terrible way to win the "Guess My Number" game

First guess:	1	Response: Too Low
Second guess:	2	Response: Too Low
Third guess:	3	Response: Too Low
and so on		
and so on		

Solution 2: Fastest way to win the "Guess My Number" game (the Binary Search algorithm)

Guess halfway between 1 and 100	First Guess: 50	Response: Too Low
Guess halfway between 51 and 100	Second Guess: 75	Response: Too High
Guess halfway between 51 and 74	Third Guess: 62	Response: Too Low
Guess halfway between 63 and 74	Fourth Guess: 68	Response: Too Low
Guess halfway between 69 and 74	Fifth Guess: 71	Response: Too High
Guess halfway between 69 and 70	Sixth Guess: 70	Response: YOU GOT IT!

Visual description of the "Guess My Number" game using the Binary Search algorithm:



Iterative Implementation

The following class contains a method that searches an array of integers for a target value using the Binary Search algorithm. Notice that the main method makes a method call to sort the array prior to calling the binarySearch method.

```
public class BinarySearch
{
   public static void main(String[] args)
   {
      int[] myArray = {23, 146, 57, 467, 69, 36, 184, 492, 100};
      int searchTarget = 57;
      insertionSort(myArray);           // Any sorting method can be used

      System.out.println(binarySearch(myArray, searchTarget));
   }

   /**
    * This method performs a Binary Search on a sorted array of integers.
    *
    * @param target the value that you are searching for
    * @param data the array you are searching
    * @return the result of the search. True if found, False if not found.
    *
    * Precondition:  The data is already sorted from smallest to largest
    */
   public static boolean binarySearch(int [] data, int target)
   {
      int low = 0;
      int high = data.length;

      while(high >= low)
      {
         int middle = (low + high) / 2;
         if (data[middle] == target)
         {
            return true;
         }
```

```
            if (data[middle] < target)
            {
              low = middle + 1;
            }
            if (data[middle] > target)
            {
               high = middle - 1;
            }
        }
        return false;
    }
}
```

```
OUTPUT
true
```

Recursive Implementation

The following is the recursive version of the Binary Search algorithm. This solution assumes the array is already sorted and it returns the index of the target.

```
public class BinarySearch
{
    public static void main(String[] args)
    {
      int[] myArray = {-230, -146, 25, 58, 179, 316, 384, 492, 500};
      int target = 25;
      int foundIndex = recursiveBinarySearch(myArray,target,0,
                                                myArray.length-1);
      if (foundIndex == -1)
        System.out.println(target + " was not found);
      else
        System.out.println(target + " was found at position " +
                           foundIndex);
}

/**
 *This method performs a Binary Search on a sorted array of integers.
 *
 *@param target the value that you are searching for
 *@param data the array you are searching
 *@return the position the target was found.
 *
 *Precondition: The data is already sorted from smallest to largest
 */
public static int recursiveBinarySearch(int[] array, int target,
                                                    int start, int end)
{
  int middle = (start + end)/2;
  if (target == array[middle])    // base case: check middle element
     return middle;
  else if (start > end) // base case: check if we've run out of elements
     return -1;                  // not found
```

```
else if (target < array[middle]) // recursive call: search start to middle
      return recursiveBinarySearch(array, target, start, middle - 1);


 else if (target > array[middle]) // recursive call: search middle to end
     return recursiveBinarySearch(array, target, middle + 1, end);
   return -1;        // not found
  }
}
```

```
OUTPUT
25 was found at position 2
```

Rapid Review

- Recursive methods are methods that call themselves.
- Choose recursion rather than looping when the problem you are trying to solve can be solved by repeatedly shrinking the problem down to a single, final simple problem.
- Recursive methods usually have a return type (rather than void) and must take a parameter.
- The re-calling of the method should bring you closer to the base case.
- Many problems can be solved without recursion; however, if done properly, recursion can be an extremely elegant way to solve a problem.
- Once the recursive process has started, the only way to end it is to satisfy the base case.
- The base case is a comparison that is done in the method. When this comparison becomes true, the method returns a value that begins the return process.
- If you write a recursive method and forget to put in a base case, or your subsequent calls to the method do not get you closer to the base case, you will probably cause infinite recursion, which will result in a stack overflow error.
- To trace a recursive call, write out each call of the method with the value of the parameter. When the base case is reached, replace each method call with its result. This will happen in reverse order.

Merge Sort

- Merge Sort uses an algorithm that repeatedly divides the data into two groups until it can divide no longer. The algorithm joins the smaller groups together and sorts them as it joins them. This process repeats until all the groups are joined to form one sorted group.
- Merge Sort uses a divide and conquer algorithm and recursion.
- Merge Sort is more efficient than Selection Sort and Insertion Sort.
- Merge Sort requires more internal memory (RAM) than Selection Sort and Insertion Sort.

Binary Search

- A Binary Search algorithm is the most efficient way to search for a target value in a sorted list.
- A Binary Search algorithm requires that the list be sorted prior to searching.
- A Binary Search algorithm eliminates half of the search list with each pass.

› Review Questions

Basic Level

1. Consider the following recursive method.

```
public int puzzle(int num)
{
   if (num <= 1)
      return 1;
   else
      return num + puzzle(num / 2);
}
```

What value is returned when `puzzle(10)` is called?

(A) `18`
(B) `15`
(C) `11`
(D) `1`
(E) Nothing is returned. Infinite recursion causes a stack overflow error.

2. Consider the following recursive method.

```
public int mystery(int k)
{
   if (k == 0)
   {
      return 1;
   }
   return 2 * k + mystery(k - 2);
}
```

What value is returned when `mystery(11)` is called?

(A) `1`
(B) `49`
(C) `73`
(D) `665280`
(E) Nothing is returned. Infinite recursion causes a stack overflow error.

3. Consider the following recursive method.

```
public int enigma(int n)
{
   if (n < 3)
      return 2;
   if (n < 5)
      return 2 + enigma(n - 1);
   return 3 + enigma(n - 2);
}
```

What value is returned when `enigma(9)` is called?

(A) `7`
(B) `10`
(C) `13`
(D) `15`
(E) `16`

4. Consider the following recursive method.

```
public void printStars(int n)
{
   if (n == 1)
   {
      System.out.println("*");
   }
   else
   {
      System.out.print("*");
      printStars(n - 1);
   }
}
```

What will be printed when `printStars(15)` is called?

(A) `***************`
(B) `*`
(C) Nothing is printed. Method will not compile. Recursive methods cannot print.
(D) Nothing is printed. Method returns successfully.
(E) Many stars are printed. Infinite recursion causes a stack overflow error.

5. Consider the following method.

```
public String weird(String s)
{
   if (s.length() >= 10)
   {
      return s;
   }
   return weird(s + s.substring(s.indexOf("lo")));
}
```

What value is returned when `weird("Hello")` is called?

(A) `"Hello"`
(B) `"Hellolo"`
(C) `"Hellolololo"`
(D) `"Hellolololololololo"`
(E) Nothing is returned. Infinite recursion causes a stack overflow error.

6. Array `arr2` has been defined and initialized as follows.

```
int[] arr2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

If the array is being searched for the number 4, which sequence of numbers is still in the area to be searched after two passes through the `while` loop of the Binary Search algorithm?

(A) `1 2 3 4 5`
(B) `2 3 4`
(C) `4 5 6`
(D) `4 5`
(E) The number has been found in the second pass. Nothing is left to be searched.

Advanced Level

Questions 7–8 refer to the following methods.

```
public int factors(int number)
{
   return factors(number, number - 1, 0);
}

public int factors(int number, int check, int count)
{
   if (number % check == 0)
   {
      count++;
   }
   check--;
   if (check == 1)
   {
      return count;
   }
   return factors(number, check, count);
}
```

7. What value is returned when `factors(10)` is called?

(A) `0`
(B) `1`
(C) `2`
(D) `3`
(E) `4`

8. Which of the following statements will execute successfully?

I. `int answer = factors(0);`
II. `int answer = factors(2);`
III. `int answer = factors(12, 2, 5);`

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

9. Consider the following recursive method.

```
public int function(int x, int y)
{
   if (x <= y)
      return x + y;
   else
      return function(x - y, y + 1) + 2;
}
```

What value is returned when `function(24, 3)` is called?

(A) `13`
(B) `21`
(C) `25`
(D) `27`
(E) Nothing is returned. Infinite recursion causes a stack overflow error.

10. Consider this recursive problem.

In elementary school, we learned that if you add up the digits of a number and get a sum of 9, then that number is divisible by 9. If your answer is less than 9, then that number is not divisible by 9. For example,

- 81 → 8 + 1 = 9, divisible by 9.
- 71 → 7 + 1 = 8, not divisible by 9.

The trouble is, if you add up the digits of a big number, you get a sum that is greater than 9, for example, 999 → 9 + 9 + 9 = 27. We can fix this by using the same trick on that sum: 27 → 2 + 7 = 9, divisible by 9.

Here's another example:

- 457829 → 4 + 5 + 7 + 8 + 2 + 9 = 35, keep going,
- 35 → 3 + 5 = 8, not divisible by 9.

If your number is big enough, you may have to repeat the process three, four, five, or even more times, but eventually, you will get a sum that is <= 9 and that will let you determine whether your number is divisible by 9.

Your client code will call method `divisible`, passing a number and expecting a `boolean` result, and `divisible` will call the recursive method `sumUp` to do the repeated addition.

Consider the class below that implements a solution to the problem.

```
public class Div9
{
   public boolean divisible (int dividend)
   {
      return sumUp(dividend) == 9;
   }

   public int sumUp(int dividend)
   {
      int sum = 0;

      // This loop adds the digits of dividend into the sum variable
      while (dividend > 0)
      {
         sum += dividend % 10;
         dividend = dividend /10;
      }

      if ( /* condition */ )
         return sum;
      else
         return sumUp( /* argument */ );
   }
}
```

Which of the following can be used to replace `/* condition */` and `/* argument */` so that `sumUp` will work as intended?

(A) condition: `sum <= 9` argument: `sum`
(B) condition: `sum = 9` argument: `dividend`
(C) condition: `sum <= 9` argument: `dividend`
(D) condition: `sum < 9` argument: `sum`
(E) condition: `sum < 9` argument: `dividend / sum`

11. Consider the following method.

```
public static int mystery (int[] array, int a)
{
   int low = 0;
   int high = array.length - 1;
   while (low <= high)
   {
      int mid = (high + low) / 2;
      if (a < array[mid])
         high = mid - 1;
      else if (a > array[mid])
         low = mid + 1;
      else
         return mid;
   }
   return -1;
}
```

The algorithm implemented by the method can best be described as:

(A) Insertion Sort
(B) Selection Sort
(C) Binary Search
(D) Merge Sort
(E) Sequential Sort