Companion Chapter 3

Recursion

What You Will Learn in This Chapter

Recursion is a powerful technique developed by computer scientists and mathematicians that has many applications in programming. Recursion can provide simple and elegant solutions to often complex problems, which makes it a must-know technique for developers. Recursive problem-solving approaches to data management and search-related tasks, among others, can provide simple solutions that enable powerful application functionality. For information systems professionals, becoming familiar with recursive techniques and approaches is highly important and will add to the set of programming tools available to the systems developer.

Specifically, this chapter will help you do the following:

1. Understand what recursion is and the nature of the problems it can solve
2. Understand how aspects of a programming language like Java can enable the application of recursive techniques
3. Explore several basic uses of recursion to further clarify its nature and the adjustments to its application that can occur when you run into problems
4. Understand and apply recursion to several very practical information systems–related problems in the realm of data and file management

Opening Scenario

While collecting requirements and planning the system and its functionality and through early project development efforts, you have worked closely with the members of your team focused on system optimization. One team member, in an all-hands status update meeting, mentioned that “speed will be a noticeable characteristic of the system as they scale this across their multiple locations.” They have pored over source code with you, looking at uses of loops, large data structures like arrays, instantiation of objects, and any wasted space in memory, among other topics. Their goal has been to look for any planned or existing code that might slow the system down horribly. You are pleased that few inefficiencies have been found, and you think the client will be very happy with the functionality and speed of the first versions of the system. The optimization team members have cautioned that one big area of concern will be the application’s ability to quickly sort and search items in its inventory: “All those point-of-sale terminals, where employees will be hand-typing item names to search and floor managers will be looking up item codes to check inventory levels—there will be a lot of searching in this application!” The optimization team members, concerned with too many calls to the database, which can add a lot of overhead, want to try to keep ranges of commonly referenced inventory in memory “to speed up both searches and any targeted analytics.”

After several discussions with the overall team, you have been asked to research and implement efficient sorting of inventory in memory and the ability to search and find inventory information fast. After some time, you discover several techniques that have been used for both sorting and searching for information. The computer science and mathematical literature states that the majority of these techniques use something called recursion to enable the speed and efficiency of the techniques that have caught your eye. “I’ll have to understand recursion as applied in a programming context before I can ensure speedy sorting and searching in a systems development

Copyright © 2024 Prospect Press. All rights reserved. For use with *Fundamentals of Java Programming for Information Systems* by Jeremy D. Ezell

context," you inform the optimization team. Luckily, it seems to be generalizable to any programming language in use for system development, including Java. One team member points you in the direction of several valuable resources, and you get to work learning as much about recursion as you can . . .

CC3.1 Understanding Recursion

Basics of recursion: Recursion is a phenomenon that can be found all around you in nature, mathematics, language, food, insect behavior, and many other areas. Mathematicians and computer scientists have found that leveraging recursive techniques can solve many problems in elegant and highly efficient ways. Wherever a problem occurs and that problem can be broken down into smaller tasks that are similar in structure and content to the larger, overall problem, a recursive approach can be used to resolve it. For example,

Problem: Imagine one hundred people standing in line. You need to sort these people by their height, from smallest to largest.

Solution: You can either sort the entire line of one hundred people all at the same time or divide the task. You might assign ten of your friends to each tackle the sorting of ten people in the line: your first friend sorts the first ten people, the second the next ten people, and so on. Finally, the groups of ten can be arranged to finish the whole sort.

Why recursion works here: The sorting of ten of those people works the same way, on the same "data," as the sorting of all one hundred at once. This means that a recursive approach can be used. Recursion is handy in "divide and conquer," where the operation that is performed on all the data can be similarly performed on a small subset of the data too.

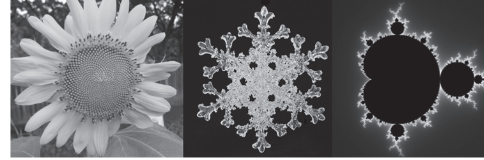

Figure CC3.1. Examples of Recursion Found in Nature and Mathematics
Sources: "Sunflower" by jon.roberts is licensed under CC BY-SA 2.0, https://www.flickr.com/photos/20712950@N00/4788459820; "File:Mandel zoom 00 mandelbrot set.jpg" by Created by Wolfgang Beyer with the program Ultra Fractal 3 is licensed under CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=321973; "Snowflake 1" by ~Brenda-Starr~ is licensed under CC BY 2.0, https://www.flickr.com/photos/37753256@N08/4420314547.

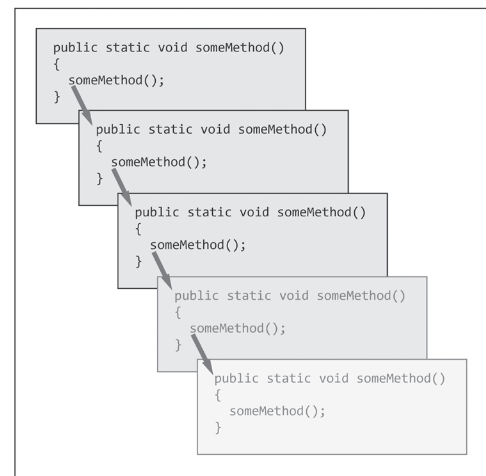

Figure CC3.2. Recursive Method Invocation Chain

In software development, there are a few things to remember about recursion:

- **Recursive techniques to problems are implemented in software by having a method invoke itself:** As you have seen in Java, you can invoke a method from anywhere in your code (as long as the method name is symbolically resolved and "visible" from your code). This works even from inside the method itself! Your method's own code can invoke itself. Consider this very simple example (we will do things the right way later):

Code Snippet CC3.1

```
public static void someMethod()
{
    // do some stuff. . .

    // Recursive self-invocation
    someMethod(); // Recursive self-invocation
}
```

Notice that the method invokes itself from within its own code body. This enables recursive approaches to problem-solving in programming (including Java!). Figure CC3.2 shows how the recursive method invocation chain looks in memory.

- **Recursion is different from looping:** Recall that a loop in Java (and any programming language) simply specifies a block of code that will be executed repeatedly. This is different from recursion, where the repetition happens due to method invocation, not code looping. In a loop, the code behaves the same way on each iteration, but with recursion, the code may act differently or act upon a different subset of data. Consider the following two methods that accept an `int` array as a parameter: one method loops while the other acts recursively (some code highlighted in **bold**):

Code Snippet CC3.2

```
// In main(). . .
int[] sortedArray = {1,2,3,4,5,6,7};
System.out.println(loopSearch(sortedArray, 3));
System.out.println(
        recursiveSearch(sortedArray,0,sortedArray.length,3));

. . .

// Static application class method
public static boolean loopSearch(int[] arr, int value)
{
    // loop
    for (int i = 0; i < arr.length; i++)
    {
        if (arr[i] == value)
            return true;
    }

    return false;
}

// Static application class method
public static boolean recursiveSearch(
        int[] arr, int start, int end, int value)
{
    int mid = ((start + end)/2);

    if (arr[mid] == value)
        return true;
    else
        if (arr[mid] < value)
            recursiveSearch(arr, start, mid, value);
        else
            recursiveSearch(arr, mid, end, value);

    return false;
}
```

Notice that no matter the length and contents of an array that you pass to `loopSearch()`, that method will act the same every single time: searching at the start until it finds (or doesn't find) the value being searched for. The contents of the array **will** cause the method `recursiveSearch()` to act differently: it does not loop; it divides the array into two halves and invokes itself to search one or the other. Loops and recursion differ greatly. The `loopSearch()` method is only invoked once, while the `recursiveSearch()` method may be invoked many times per array.

- **Logic must be used to determine when a method will stop recursively invoking itself:** In Java, if a method recursively invokes itself, and that call then invokes itself again and so on, you will eventually run out of memory in the frame stack area of memory. In fact, if the code for `someMethod()` (from earlier) is executed, the following error occurs:

```
Exception in thread "main" java.lang.StackOverflowError
```

A stack overflow (this term might be familiar to you if you frequently search for code examples online!) error occurs when the frame stack has so many method frames within (all from method invocations) that it runs out of its allocated memory. There's lots of room in memory but not an infinite amount of it. Frequently, failing to include logic to stop a recursive method invocation "chain" will cause this. You will hear this stopping logic called a "base case," "anchor case," and "stopping case" in the programming literature. Consider the following adjustment to our `someMethod()` method from earlier that introduces a base case into the recursion chain:

Code Snippet CC3.3

```
// in main()
someMethod(1);
. . .
public static void someMethod(int num)
{
    // Stopping Case
    if (num == 10)
        return;

    // Recursive self-invocation
    someMethod(++num);
    System.out.print(num + " ");
}
```

When this executes, the following prints to the console:

```
10 9 8 7 6 5 4 3 2
```

The stopping case in a recursive method is usually implemented with an `if` or an `if…else` decision statement.

- **(Most) all problems that can be solved in an iterative fashion can be solved with a recursive approach:** In the prior example, what did the recursion "chain" of method invocation do? It simply printed the numbers one through ten in reverse. Can this be done with a simple `for` loop? Of course:

Code Snippet CC3.4

```
for (int i=10; i>0; i--)
{
    System.out.print(i + " ");
}
```

Notice that the recursive approach printed the same numbers as this traditional, more iterative `for` loop approach. But even though you can solve iterative problems with recursion, it does not always mean you **should**. When considering using recursion, consider the following two questions:

  - **Is the recursion version of the solution more or less elegant?** Your recursion code may solve the problem, but it may overcomplicate the code, its maintenance, and the ability of other programmers to understand it.
  - **Is the recursion version of the solution more or less computing intensive?** In the code example, the use of a `for` loop takes up much less memory (less "overhead") than the recursive approach. Recall that each invocation of a method requires space to be allocated in memory, much more than with the use of a simple `for` loop. In applications that may scale very large across many users/systems, which approach would be faster? Sometimes recursion can speed up a task dramatically, and other times, it can slow it down or even stop the solution from being fully carried out.

- **Recursive techniques work best when the problem at hand can be subdivided into smaller tasks that resemble the "whole":** This was stated earlier in this section. Notice with the number print example that the printing of a number sequence can be handled all at once with one `for` loop or "piecemeal" by each method invocation. The subtask looks like the whole task. When this symmetry exists, a recursive approach may work. Consider examples like sorting a line of people (discussed earlier), searching for one document in a file drawer containing thousands, or counting the number of files stored within a hierarchy of folders on a hard drive or cloud location.

Simple examples of recursive approaches to solving problems in Java: Once you have identified a programming problem that might benefit from a recursive technique, you will need to implement the code to apply it. Recursive programming requires you to adapt your thinking away from a brute-force iterative approach to a more segmented, divide-and-conquer one. First, we can examine a few simple examples of recursive programming techniques and discuss whether recursion was a benefit or not in each. For a few of these, both the iterative and recursive approaches will be covered.

- **Printing a number range from `0` up to `num`:**

Iterative approach: Printing numbers from `0` to `num` is relatively straightforward using a `for` loop:

Code Snippet CC3.5

```
int num = 7;
for (int i=0;i<num;i++)
{
    System.out.print(i + " ");
}
System.out.println();
```

Notice that the printing of each individual number is handled on each iteration of the loop. The loop knows when to stop because the variable `num` is used in the loop-until condition.

Recursive approach: In order to use a recursive approach, keep a few things in mind:

- A method will be needed that can invoke (i.e., call) itself.
- Each number in this sequence will be printed once within a particular invocation of the method.
- A "base case" is needed so that the invocation chain stops, and the methods can all finish and leave memory, bringing execution back to the `main()` method.
- Because the method is calling itself, each invocation needs to be able to pass the prior `num` printed to the next invocation so that it knows what to print next.

You can assume that the number `num` will be passed to the method from whatever code invokes it to start the recursion chain. Your first attempt might look something like this:

Code Snippet CC3.6

```
public static void printIncreaseCount(int num)
{
    System.out.print(num + " ");
    printIncreaseCount(++num);
}
```

The method does invoke itself, which is the first step in applying recursion to the problem. The current value of `num` is printed, and then a pre-increment operator is used to increase it before passing it to the next invocation of `printIncreaseCount()`. The problem with this attempt is that a base case is missing: How do you stop the invocation chain from going into infinity (or a `StackOverflowError`)? Another problem here is that `num` is the value to stop printing at, not start at. A simple solution could look like the following (changes in **bold**):

Code Snippet CC3.7

```
public static void printIncreaseCount(int start, int num)
{
    if (start > num)
        return;

    System.out.print(start + " ");
    printIncreaseCount(++start, num);
}
```

Notice that each invocation of the method passes to the next the value of `num`, the value to stop at. The variable `start` is also incremented after printing it. The base case here is the check to see if the value of `start` has exceeded `num`. If so, you simply call `return` (for a `void` method, return can be called by itself, causing execution to leave the method).

This logic can be additionally simplified:

Code Snippet CC3.8

```
public static void printIncreaseCount(int start, int num)
{
    if (start <= num)
    {
        System.out.print(start + " ");
        printIncreaseCount(++start, num);
    }
}
```

Instead of explicitly calling `return`, the `if` statement's logic is flipped and only continues the recursive invocation if `start` is less than or equal to `num`. Invoking this from `main()` with the following:

Code Snippet CC3.9

```
printIncreaseCount(0, 7);
```

will produce the following output:

```
0 1 2 3 4 5 6 7
```

This works! But you are still not leveraging the full power that recursion provides. This implementation has to pass both the number to stop at and the number to start at. The original intent was to implement a recursive solution that prints numbers up from `0` always. Requiring the developer to pass in a `0` as the first parameter for each call is frustrating and a decrease in the elegance of the solution. Consider the following final version of this recursive method:

Code Snippet CC3.10

```
public static void printIncreaseCount(int num)
{
    if (num > 0)
        printIncreaseCount(num - 1);

    System.out.print(num + " ");
}
```

From your studies in chapter 6, you might remember a few characteristics of methods and primitive data types as parameters:

- Primitive data-type parameters are local to the method.
- Primitive data-type parameters receive a copy of a value from the argument passed to the method during the invocation.

Recall also from figure CC3.2 that recursive methods form a chain. Each prior method on that chain waits for the one it invoked to finish before it can finish (this is true of normal invocation between methods as well). With the number print, you leverage this fact in the following ways:

- The first invocation prints the largest number, the next prints the next smallest, and so on.
- The call to `.print()` only occurs after the recursive call to the next version of `printIncreaseCount()`.
- When the value of num reaches `0`, the base case has been "hit," the recursive call does not happen further, and `0` is printed first to the console.
- The last method in the invocation chain finishes and leaves memory, and execution returns to the prior method in the chain, which executes its `.print()` and so forth back up the chain. See figure CC3.3 for an illustration of this.

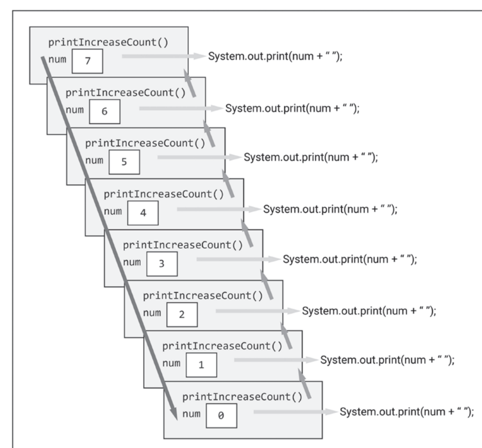

Figure CC3.3. Illustration of the `printIncreaseCount()` Chain

Here, recursion works through the sequence in reverse, but printing starts with the last method called.

Displaying the Individual Digits in a Large Number

Iterative approach: The task here is to extract each individual digit from a larger number and print each number separately. Converting the number to a `String` and printing each character could work, but the use of division and the modulo operator will work better to accomplish this task:

Code Snippet CC3.11

```
long num = 186987; // Large number
long remainder;
String result = "";
while (num > 0)
{
    remainder = num % 10;
    num = (num / 10);
    result += remainder + "";
}
for (int i=result.length() - 1; i>=0; i--)
{
    System.out.print(result.charAt(i) + " ");
}
System.out.println();
```

Notice the approach: First, the modulo operator is used to determine the remainder when dividing the whole number by `10`. This yields a remainder of `7`. The remainder is saved, and then the number is again divided by `10`, and that quotient is saved, "shaving off" the last number. This is iteratively repeated until the quotient is `0` with a remainder. Because we extract the last digit first, and the digits are concatenated into a `String`, we print the characters of the `String` in reverse.

Recursive approach: Notice that this problem fits well with a recursive approach: the extraction of each digit is the same as any other (subtask same as the overall task), and as with the number print example earlier, the first digit is extracted at the end of the task, meaning printing of it should happen there too. Consider this recursive approach to this problem:

Code Snippet CC3.12

```
public static void displayNumberDigits(long num)
{
    if (num > 10)
        displayNumberDigits(num / 10);

    System.out.print((num % 10) + " ");
}
```

With this solution, the method invokes itself and divides `num` by `10`, "chopping off" the last digit and sending a copy of the smaller num value to the next invocation. At the base case, the parameter value falls below `10`, and the invocation chain stops. Then the next line of code after the call can execute, the method ends, and execution returns to the prior up the chain. The printing of the digits happens in the natural order, but it was processed starting with the ending digit—an elegant solution that uses the nature of a recursive algorithm to its advantage! Notice the simpler logic with the recursive approach versus the iterative one.

Computing the Factorial of a Number

Finding the factorial of a number is a common example of recursive programming. As a reminder, the factorial of a number consists of taking that number and multiplying it against all the numbers behind it to `1` (with the special case of `0!` equal to `1`). Often this is indicated with a number followed by an exclamation point, like "5!," which would be pronounced "five factorial." Mathematically, this would break down into multiplication like so:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Iterative approach: The iterative approach here is fairly straightforward: the use of a loop and a variable to hold the growing product are all that are needed:

Code Snippet CC3.13

```
long factorial = 5;
for (long i=factorial - 1; i>0; i--)
{
    factorial *= i;
}
System.out.println(factorial);
```

Notice that the next lower number from the original factorial is used as the first value of `i` in the `for` loop.

Recursive approach: Since you now know that recursion involves method self-invocation, you can allow each successive invocation to handle the next multiplication in the factorial series. In this approach, the method will return back a `long` that will be used by the prior invocation:

Code Snippet CC3.14

```
public static long findFactorial(long num)
{
    if (num <= 1)
        return 1;
    else
        return num * findFactorial(num - 1);
}
```

Notice the base case here will be the special factorial results of `1!` or `0!`. The first invocation of the method will return to the calling code `num * (num - 1)`. The second invocation receives `(num – 1)` and returns back `(num – 1) * (num – 2)`, and so on, building the factorial chain, each subtask similar to the whole. The trade-off here is elegance of the solution versus memory overhead of using a recursive approach.

Calculating a Specific Fibonacci Number

Another common example used to help understand recursive programming is writing code to generate the value of a specific Fibonacci number. Fibonacci numbers follow a certain ever-increasing sequence, and this sequence occurs in nature (tree leaves and branching, ice patterns, flower structure, etc.) and has applications to problems in mathematics, data management, and finance, among other areas.* The Fibonacci sequence can be seen in action in figure CC3.1 in the structure of the sunflower, the way the branches of a snowflake grow, and the fractal Mandelbrot patterns.† The value of a specific Fibonacci number is the sum of the two Fibonacci numbers behind it and so on in the series. In math terms, the formula for this would be the following:

```
Fn = F(n - 1) + F(n - 2)
```

And the first fifteen Fibonacci numbers in the series are listed in table CC3.1:

Table CC3.1. First Fifteen Numbers in the Fibonacci Sequence

F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

The challenge here is that the fifth Fibonacci number—F_4, for example—cannot be found until you know what the values for F_3 and F_2 are, and those cannot be known until the ones behind them are calculated, and so on. The value for F_0 is always 0 and F_1 is always 1, so these help to start the calculation of a sequence. Since each number is calculated the same way, this lends itself to a recursive approach very nicely:

* https://en.wikipedia.org/wiki/Fibonacci_sequence.
† https://en.wikipedia.org/wiki/Mandelbrot_set.

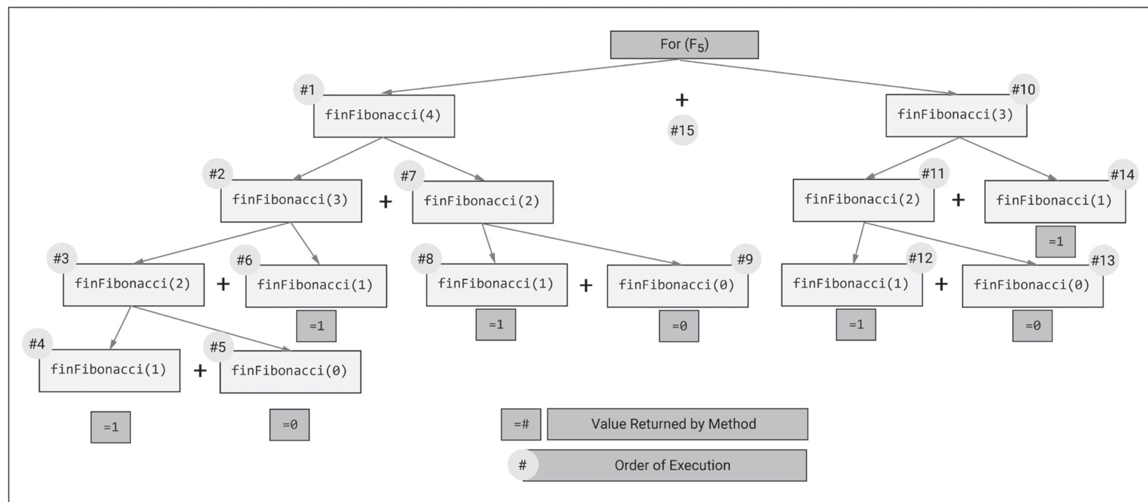

Figure CC3.4. Fibonacci Recursive Call Branching for F(5)

Code Snippet CC3.15

```
public static long findFibonacci(long num)
{
    if (num == 0)
        return 0;
    else if (num == 1)
        return 1;
    else
        return findFibonacci(num - 1) + findFibonacci(num - 2);
}
```

The recursive approach here is slightly different than in the prior examples. Notice here that the method `findFibonacci()` is invoked **twice** from within each recursive method invocation. Earlier the invocations were chained, whereas here the invocations branch out into two "paths." Each invocation will call two other instances of the `findFibonacci()` method, passing a parameter value to it. At the end of these branching chains, the value of num will reach `0` or `1`, causing the base case to trigger and return `0` or `1`, respectively, for those values. Execution will return up the branches, adding all the `1` values until the specific Fibonacci value requested has been summed. Figure CC3.4 shows this branching for the calculation of F_5, whose value is summed up to be `5`.

You can invoke the method to calculate the first twenty Fibonacci numbers easily with the following code in a `main()` method:

Code Snippet CC3.16

```
for (int i=0; i <= 20; i++)
    System.out.println(
            "F(" + i + "): "
                    + findFibonacci(i));
```

This code produces the following output:

```
F(0): 0
F(1): 1
F(2): 1
F(3): 2
F(4): 3
F(5): 5
F(6): 8
F(7): 13
F(8): 21
```

```
F(9): 34
F(10): 55
F(11): 89
F(12): 144
F(13): 233
F(14): 377
F(15): 610
F(16): 987
F(17): 1597
F(18): 2584
F(19): 4181
F(20): 6765
```

In this case, recursive behavior of the code lends an elegance to the solution that makes implementation and usage easy.

Challenge: Change the `for` loop so that it finds up to the fiftieth Fibonacci number (F_{50}). Notice how the calculations slow down a great deal the closer to the fiftieth Fibonacci number you get. This recursive branching can get very complicated even though it is only simple addition!

```
. . .
. . .
F(42): 267914296
F(43): 433494437
F(44): 701408733
F(45): 1134903170
F(46): 1836311903
F(47): 2971215073
F(48): 4807526976
F(49): 7778742049
F(50): 12586269025  // Takes a really long time. . . 12+ Billion!
```

When Recursion Can Hinder Efforts: Approximating Pi (π)

Because of the sequential invocation chaining that occurs with recursion, most iterative solutions to problems can be converted to a recursive approach. But this does not mean that they **should** be. Sometimes the overhead involved with recursion can hinder or even prevent a full solution. Consider the task of approximating the value of pi (π). Pi is equal to the circumference of a circle divided by the diameter. Since measurements of these are never perfect in the real world, several mathematical techniques have been developed to approximate pi, one of these being the Leibniz formula, which looks like this:*

$\pi = 4 * ((1/1) - (1/3) + (1/5) - (1/7) + (1/9) - \ldots (1/n))$

This formula is commonly used in programming examples because it tests your ability to use expressions, operators, boolean logic, and loops to arrive at the solution. This formula is said to approximate pi because it comes close but is never exactly right. The further out you calculate the fractions in the formula (the bigger the *n*), the more digits of pi you can extract. But it takes a lot of terms to extract even a few digits. Pi's first ten digits are as follows:

`3.1415926535`

Iterative approach: The following method uses a nonrecursive, iterative approach to approximate pi using the Leibniz formula. This approach has limits too, particularly the precision limit of the `double` data type. A better use might be the `BigDecimal` class or the implementation of a class of your own so that a (reasonably) unlimited number of decimals can be calculated. The method looks like this:

Code Snippet CC3.17

```
public static double piTerms(long numberOfTerms)
{
    double pi = 1;
```

* https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80.

```
    double denom = 3;
    boolean subtract = true;

    for (int i=0;i<numberOfTerms; i++)
    {
        if (subtract)
        {
            pi -= (1.0 / denom);
            denom += 2;
            subtract = false;
        }
        else
        {
            pi += (1.0 / denom);
            denom += 2;
            subtract = true;
        }
    }
    return pi;
}
```

Notice that as with the Leibniz formula, an `if…else` allows for the "flip-flop" of a subtraction or addition for each successive term. The parameter `numberOfTerms` determines how many fractions to sum up. Here are some separate runs of this method to show you how poorly and slowly it can approximate pi (correct terms in **bold**):

```
System.out.println(piTerms(10));        // Ten Terms
System.out.println(piTerms(100));       // One Hundred
System.out.println(piTerms(1000));      // One Thousand
System.out.println(piTerms(10000));     // One Hundred Thousand
System.out.println(piTerms(1000000));   // One Million
. . .
3.232315809405594
3.1514934010709914
3.1425916543395442
3.1416926435905346
3.1415936535887745
```

Notice that it takes over one million terms summed up just to extract five decimal places in pi!

Recursive approach: There are several ways to handle this algorithm recursively. The "flip-flop" of the sign can be handled by simply multiplying the return from the next recursive call by `-1` (subtracting a negative yields a positive, etc.). As with prior examples, we start at the end of the series (with the largest value of *n*) and sum backward through the series. The base case of the parameter value reaching `0` ends the invocation chain, and the methods return, adding up the returned value as each closes:

Code Snippet CC3.18

```
public static double recursivePiTerms(long numberOfTerms)
{
    if (numberOfTerms != 0)
    {
         return (1.0 / ((2.0 * numberOfTerms) + 1))
                + (-1 * recursivePiTerms(numberOfTerms - 1));
    }
    else
         return 1.0; // Base Case
}
```

The multiplication by 4 is done in the calling code to simplify the method (otherwise, we would have to pass a value indicating how far into the series we are, multiplying the sum by 4 before we return to the calling code). Testing this out, it seems to work well!

```
System.out.println(4 * recursivePiTerms(100));
System.out.println(4 * recursivePiTerms(1000));
. . .
3.1514934010709914
3.1425916543395442
```

Recursing one hundred terms yielded one decimal place! What about one thousand? That yielded two places. When you try to run this with ten thousand terms, an issue occurs:

```
System.out.println(4 * recursivePiTerms(10000));
. . .
Exception in thread "main" java.lang.StackOverflowError
```

Remember that recursion adds memory overhead from method invocation. Too many methods in memory will fill up the frame stack and lead to a `StackOverflowError`. This is a good lesson: **just because** an iterative solution can be converted to a recursive one **does not mean it should**. Will the overhead prevent the new solution from completing? In this case, it did. Because the Leibniz formula is so inefficient, it takes a lot of terms to extract pi digits. Because each term requires a method call in a recursive solution, there are only so many that can be calculated. So in this case, we need to stick with the iterative solution or investigate other mathematical approximations of pi that might recurse better (research that this author will leave up to you!).

SUMMARY POINTS

- Using recursive techniques in programming allows for complex processes to occur with fairly simple code implementations.
- Recursion in Java is enabled by implementing a method so that it invokes itself from within its own code body.
- A base case must be implemented in the logic of the recursive method so that recursive calls stop and method invocations begin ending, shortening the method invocation "chain" back to the original method execution.
- All iterative problems in programming can be converted into a recursive-based solution, but many should not, as in many cases, it could hinder both the simplicity of the logic and the completion of the solution.
- Recursion works best with smaller chunks of a task and can be subdivided into portions that resemble the efforts of the whole task.

QUICK PROBLEMS

1. **Coding:** Write a recursive method that prints numbers in a sequence, starting from a high number down to zero but skipping each even number. Ensure a working base case that stops the invocation chain.
2. **Think:** Why is the use of a method critical to enabling recursive approaches to problem-solving in programming?
3. **Coding:** Write a recursive method that accepts a String phrase and prints that phrase on each invocation, leaving off one additional letter on the end of the phrase on each print. Ensure a working base case that stops the invocation chain.

CC3.2 Practical Business Uses of Recursion in Java

Discussions and examples of recursion in programming can quickly wander into very interesting discussions involving topics in computer science, mathematics, and algorithmic efficiency. Though these are very important topics, they are beyond the scope of this book's goal: to examine Java programming concepts relevant to information systems professionals (that's you!). You are encouraged to search and explore discussions on those topics regarding recursion as you are interested, particularly on how efficient various recursion techniques are. In the following section, you will see several practical applications of recursion to programming problems that information systems developers might face. Keep in mind that these techniques have all been adapted from the valuable work done by the community of computer scientists, mathematicians, and others.

Figure CC3.5. Line of People Waiting to Be Recursively Sorted
Source: "Queue" by gadl is licensed under CC BY-SA 2.0, https://www.flickr.com/photos/24183489@N00/89650415.

Using Recursion to Print the Contents of a Directory and Its Subdirectories

When building an information system, you will have frequent need for the software to be able to open from and save to a directory (on either a hard drive or a cloud location mapped as a drive to the local machine). Similarly, you might need to display the contents and structure of a directory and all the files and subdirectories within it to help the user navigate files and choose what file to open and where to save.

Think about the overall task: displaying files in a directory. This is the same whether or not you are in a top-level directory or a subdirectory eight folders deep. In this case, the subtask is similar to the overall task, making this a great problem to solve using recursive techniques.

Consider the directory structure shown in figure CC3.6 with files and subdirectories found within a Java project folder (this could be in NetBeans, Eclipse, jGRASP, or other environments):

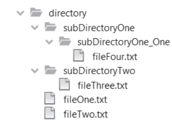

Figure CC3.6. A Local Directory Structure in a Java Project

The directory and its contents do not have to be local to a Java project per se: they can be located anywhere on a hard drive or mapped drive. For this example, the folders are part of a Java project directory that will simplify the example code. How can the contents be recursively listed? Consider the following algorithm:

- In a method, list all the items in the top-level folder.
  - If the item is a file, print its name.
  - If the item is a subdirectory, recursively invoke the method, passing it to this directory.

The recursive call will descend one folder down each time a directory is found, listing files and directories stored within and recursively "crawling" down those folders as well until no more are found. First, here is the code that implements the recursive approach (*Note:* you will need to import java.io.* in order to use the File class and its methods; the recursive method invocation is in **bold**):

Code Snippet CC3.19

```
public static void recursiveDirectoryPrint(
        File fileToExplore, int level)
{
    if (fileToExplore.isFile())
    {
        for (int i=0;i<level;i++)
            System.out.print("   ");
        System.out.println(fileToExplore.getName());
    }
    else if (fileToExplore.isDirectory())
    {
        if (level > 1)
```

```
            for (int i=0;i<level;i++)
                System.out.print("   ");

        System.out.println(fileToExplore.getName());

        level += 1;

        for (File f: fileToExplore.listFiles())
        {
            recursiveDirectoryPrint(f, level);
        }
    }
}
```

Notice that the parameter `level` is used merely to help give visual structure to the directory structure output that prints. Invoking this method on the local Java project directory folder looks like the following:

Code Snippet CC3.20

```
// Recursive Approach
File testFile = new File("./directory");
recursiveDirectoryPrint(testFile, 1);
```

This produces the following output:

```
directory
      fileOne.txt
      fileTwo.txt
      subDirectoryOne
         subDirectoryOne_One
            fileFour.txt
      subDirectoryTwo
         fileThree.txt
```

Figure CC3.7 shows the visual "crawl" of this recursive algorithm through the directory structure in the folder. Notice that the "crawl" is similar in behavior to that shown in the Fibonacci example in figure CC3.4.

Searching an Array Recursively for a Specific Value Using Binary Search

Recall from the discussion in chapter 7 that an array is a series of elements, and this series is mapped using index notation from `0` to `n`, `n` being the quantity of "things" in the array (values, object references, etc.). A common task that developers in information systems often face is searching for an array or series of data for a specific value. Often this occurs with a database search of a table (using an SQL query). If values are stored in a memory-located data structure like an array or `ArrayList<>`, then knowing how to search that array efficiently is a valuable skill for the information systems developer. This is particularly important when you have created objects of custom data types for your application and need to perform search actions upon arrays of them. In these cases, any built-in search actions may not be guaranteed to work properly, if at all. Implementing your own search can ensure they do.

Iterative approach: Consider the following iterative approach to searching an int array for a particular value. Note the logic for whether the value was found or not:

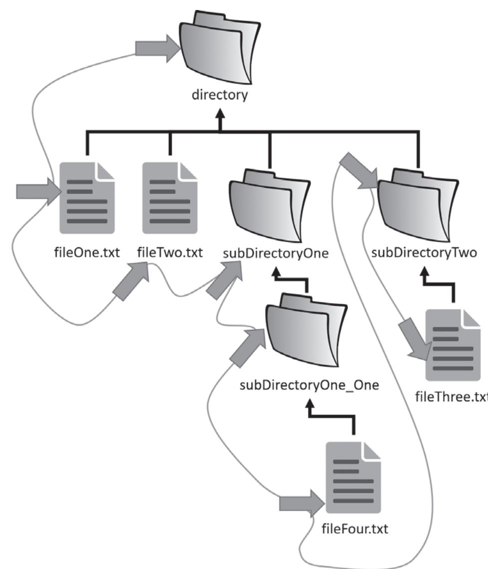

Figure CC3.7. Visual "Crawl" to Recursively Print the Directory Structure

Code Snippet CC3.21

```
int[] array = {1, 3, 4, 6, 10, 16, 24, 34, 78, 101};
int valueToFind = 78; // Example value to search for
int i = 0; // declaration and initialization of i

for (; i<array.length; i++) // i declared outside for loop
{
    if (array[i] == valueToFind)
        break;
}
if (i < array.length)
    System.out.println("Value found at index " + i);
else
    System.out.println("Value NOT found!");
```

The loop control variable `i` is declared outside the `for` loop so that it can be used afterward (scope rules). For an array this small, this is a fairly efficient search. If this search was being conducted thousands of times a second, the inefficiencies would rise. Every element of the array has to be compared to the `valueToFind` every time a search is performed until the element is found. If the array is much larger than this, say several thousand elements in quantity, and the search is being performed several thousand times a second, things get slower.

Recursive approach: Using recursion, computer scientists have developed a technique (among many others) for efficiently searching an array of values for a particular one. The binary search uses a "divide-and-conquer" approach to finding the element being searched for while minimizing (as much as possible) the number of comparisons between array elements and the `valueToFind`. Again, chapter 7 demonstrates the use of the `binarySearch()` method in the `Arrays` class. As developers working within information systems contexts, you may need to write your own version of this method for custom data types you work with. The binary search technique is also very dependent on recursion for its efficiency. Since searching a subpart of the array is the same as searching all of it (smaller task similar to the overall), recursion is highly applicable here. A binary search works in the following way:

- Use an already sorted array (this is a **critical prerequisite**).
- Find the middle element of the array. Compare the `valueToFind` to the middle element.
  - If the `valueToFind` is smaller than the middle element, continue the search in the left half of the array.
  - If the `valueToFind` is larger than the middle element, continue the search in the right half of the array.
- Repeat this middle element comparison with the appropriate half of the array, and continue subdividing until the element is found (or not).

Figure CC3.8. Visual Demonstration of the Binary Search Technique

Using the array from earlier, figure CC3.8 demonstrates this binary search algorithm visually.

For binary search to work correctly, it is critical that the array used **has already been sorted**. You will look at an efficient sorting technique later in this chapter. Since this algorithm divides and conquers the array, it eliminates half the elements under consideration at the start of each search and continues dividing in half until it reaches the value being searched for. As you have seen, recursion can help subdivide a task down until a base case is reached, which is exactly what we need here. The following is one implementation of the binary search algorithm for an int array of values (the recursive invocations to the method are highlighted in **bold**):

Code Snippet CC3.22

```
public static boolean binarySearch(
        int[] arrayToSearch,
        int valueToFind,
        int start, int end)
{
    int indexMidPoint = -1; // Value Will Change

    System.out.println(
            "Searching from indexes " + start
                    + " to " + end);

    if (start == end)
    {
        if (arrayToSearch[start] == valueToFind)
            return true;
        else
            return false;
    }

    indexMidPoint = ((start + end) / 2);

    if (arrayToSearch[indexMidPoint] == valueToFind)
        return true;
    else if (valueToFind < arrayToSearch[indexMidPoint] )
        return binarySearch(
                arrayToSearch, valueToFind,
                start, indexMidPoint - 1);
    else
        return binarySearch(
                arrayToSearch, valueToFind,
                indexMidPoint + 1, end);
}
```

A few things to point out about this implementation:

- It assumes that the array reference it receives is for an **already sorted** array.
- The first selection statement tests if the array range being searched only contains one element (if the start and end are equal). In that case, a simple comparison is made. This will occur at the end of the recursive invocation chain.
- Notice that if the `valueToFind` is less than the midpoint value, the range is set from the first element in the array to the element just before the midpoint, and a recursive call is made with that range. The next invocation of the method will find the midpoint of that smaller range and subdivide some more.

Using the recursive binary search method upon the array exampled earlier yields the following:

Code Snippet CC3.23

```
// Binary Search
int[] array = {1, 3, 4, 6, 10, 16, 24, 34, 78, 101};
int valueToFind = 78;
System.out.println("Is " + valueToFind + " in the array?:");
System.out.print(
        binarySearch(
                array,
                valueToFind,
                0, array.length - 1));
```

The output from this would be the following:

```
Is 78 in the array?:
Searching from indexes 0 to 9
Searching from indexes 5 to 9
```

```
Searching from indexes 8 to 9
true
```

Notice the printing of the indexes. On the first invocation, all ten elements were considered. Since the midpoint is chosen with the formula `(start + end) / 2`, most likely the `int` index element `4` (value `10`) was chosen as the midpoint. Since `78 > 10`, the array was partitioned into two halves, and only the second, larger half was considered. The recursive invocation passed this new range into the next method call, and the task was performed again until the value was found.

Sorting an Array Using the Recursive QuickSort Technique

Since many search techniques like binary search require an array that is already sorted, understanding **how to sort** is equally important. For information systems professionals often working with custom data types/classes, the argument is the same: the built-in sorting methods available in the `Arrays` class (and other places) may not work correctly or at all with an array or series of your data types. You may need to implement your own sort algorithm. Luckily, there are many to choose from (and you are invited to research the others we do not discuss): bubble sort, merge sort, selection sort, and **many** others. The QuickSort technique, like binary search, is an algorithm that benefits from the power of recursion and is a good sort technique to explore in this chapter. While incredibly efficient, the quick sort algorithm can be a bit tricky to understand (but not impossible). In plain language, **here is how it works** (an example array is printed here so that you can have a helpful visual reference while reading the algorithm):

Array used for QuickSort algorithm:

```
{3, 8, 10, 2, 7, 4, 11, 21, 6}
```

- Given an array of elements, pick one element to serve as a **Pivot** element (there are several opinions on how to do this effectively—the element right in / near the middle works well).
- Swap the **Pivot** element with the **very last element** in the array range.
- Choose the **first element** in the array as the **LeftPointer**. Choose the **second-to-last** element as the **RightPointer**.
- **Walk** the **LeftPointer** to the **right** along the array until you encounter an element that is **larger than the value of the Pivot**.
- **Walk** the **RightPointer** to the **left** along the array until you encounter an element that is **smaller than the value of the Pivot**.
- Swap the element values of **LeftPointer** and **RightPointer**, but keep the pointers at the index positions they are at.
  - Continue the walk of **LeftPointer** to the **right** and **RightPointer** to the **left**, swapping numbers as stated **until the LeftPointer is farther to the right than RightPointer.**
- **Swap** the value of the elements at **Pivot** and **LeftPointer** only. Change the **Pivot** index accordingly.
- Using **recursion**, repeat this entire algorithm on the **elements to the left of the Pivot element** and to the **elements to the right of the Pivot element**. **Do not include the Pivot** element, as it is now sorted in its appropriate position.

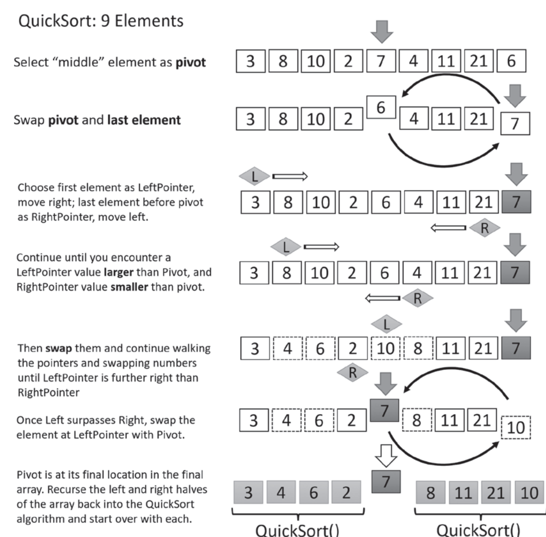

Figure CC3.9. Visual Demonstration of the QuickSort Algorithm

Figure CC3.9 gives a visual demonstration of how the QuickSort algorithm works on the earlier example array.

The algorithm is a bit complex, as is the code needed to implement it. Based on what you research, there are several "flavors" of QuickSort, and many professionals will implement it in slightly different ways. The code for the recursive method that implements the algorithm is as follows (with comments and the recursive calls in **bold**):

Code Snippet CC3.24

```
public static void recursiveQuickSort(
        int[] numArray,
        int start, int end)
{
    // Assumption: "end" is the last index,
    //  not the array length.

    // Very simple pivot choice
    int pivotIndex = (start + end) / 2;
    int leftCompareIndex;
    int rightCompareIndex;
    int swapHolder;

    if (end < start || end == start)
        return;

    // If the range is size two, simply compare
    //  and swap if necessary:
    if (end - start == 1)
    {
        if (numArray[start] >= numArray[end])
        {
            swapHolder = numArray[start];
            numArray[start] = numArray[end];
            numArray[end] = swapHolder;
            return;
        }
    }

    // Swap the pivot number and the last number
    swapHolder = numArray[end];
    numArray[end] = numArray[pivotIndex];
    numArray[pivotIndex] = swapHolder;

    // Change the pivotIndex
    pivotIndex = end;

    // Set the comparison indexes
    leftCompareIndex = start;
    rightCompareIndex = pivotIndex - 1;

    // Begin the left and right comparisons to pivot.
    while (leftCompareIndex < rightCompareIndex)
    {
       while (numArray[leftCompareIndex] <= numArray[pivotIndex]
               && (leftCompareIndex < end))
       {
           // "walk" right
           leftCompareIndex++;
       }

       while (numArray[rightCompareIndex] >= numArray[pivotIndex]
               && (rightCompareIndex > 0))
       {
           // "walk" left
           rightCompareIndex--;
       }

       // If the left comparison passed the right, we're done.
```

```
        // This skips the swap.
        if (leftCompareIndex > rightCompareIndex)
            break;

        // If the while loops have stopped we can swap numbers
        swapHolder = numArray[rightCompareIndex];
        numArray[rightCompareIndex] = numArray[leftCompareIndex];
        numArray[leftCompareIndex] = swapHolder;

        // The search can continue. . .
    }

    // If the while loop is done, we've completed the comparisons.

    // Swap the pivot and the leftCompare number
    swapHolder = numArray[leftCompareIndex];
    numArray[leftCompareIndex] = numArray[pivotIndex];
    numArray[pivotIndex] = swapHolder;

    // Update the pivotIndex
    pivotIndex = leftCompareIndex;

    // Recurse each half of the array around and excluding the pivot
    recursiveQuickSort(numArray, start, pivotIndex - 1); // left
    recursiveQuickSort(numArray, pivotIndex + 1, end); // right
}
```

Notice that in the recursive invocation, only the range to the left of the now-sorted pivot is passed into one invocation and the range to its right into the second invocation of the method. The base case here is when the invocation chain examines just one or two elements:

- If two elements, a simple comparison and swap are performed.
- If only one element, the method simply returns with no actions taken.

An example invocation on several test arrays would look like the following (*Note:* a method called `arrayPrint()` is also invoked, written by the author, that simply prints the contents of an `int[]` array with bracket decorations around each element for ease of reading; see if you can implement this additional method on your own):

Code Snippet CC3.25

```
int[] array = {3,8,10,2,7,4,11,21,6};
int[] array2 = {89,5,34,65,22,11,9,3,90,103,45};
int[] array3 = {2, 3, 4, 22, 4, 19, 109};

arrayPrint(array);
recursiveQuickSort(array, 0, array.length – 1);
arrayPrint(array);
System.out.println();

arrayPrint(array2);
recursiveQuickSort(array2, 0, array2.length – 1);
arrayPrint(array2);
System.out.println();

arrayPrint(array3);
recursiveQuickSort(array3, 0, array3.length – 1);
arrayPrint(array3);
```

This produces the following output (with sorted arrays highlighted here in **bold**):

```
[3][8][10][2][7][4][11][21][6]
```

```
[2][3][4][6][7][8][10][11][21]

[89][5][34][65][22][11][9][3][90][103][45]
[3][5][9][11][22][34][45][65][89][90][103]

[2][3][4][22][4][19][109]
[2][3][4][4][19][22][109]
```

The algorithm is effective even when there are duplicate values or when the largest and smallest values are already at the ends of the array. Like its name implies, the QuickSort algorithm is very efficient in terms of producing a sorted array quickly without a lot of unnecessary looping or value comparisons. QuickSort is the implemented sort technique for many of the built-in sorting methods and code that you might call in Java and other languages as well.

Summary

In this chapter, you have learned about the basics of the recursive approach to problem-solving in programming. The ability of methods to call, or invoke, themselves is a key to enabling recursive behavior in applications. You have also learned the importance of implementing a base case in a recursive method so that the invocation chain eventually ends. Additionally, you have explored some basic and more practical uses of recursion to implement very elegant and effective problems often faced by information systems professionals attempting to programmatically manage data in an application. There are many, many more types of problems where recursion can be effectively applied beyond those listed in this chapter. Particularly, visual and graphics display applications (such as the Mandelbrot set/fractals) are commonly discussed uses that you will find in your research. You are encouraged to explore other problems that recursion can be applied to as a part of rounding out your overall professional development knowledge.

Practice Problems

Terminology

Match the following terms from the chapter with their most appropriate definition:

Term	Definition
1. Recursion	a. Problem-solving technique in programming that uses loops and selection statement logic to sequentially perform a subtask.
2. Invocation chain	b. The element that divides the work in the QuickSort algorithm.
3. Base case	c. Algorithm that implements a "divide-and-conquer" technique to sorting a series of values.
4. Iterative approach	d. The connection between iterations of a method that has recursively invoked itself to perform a task.
5. Overhead	e. The element that divides the work in a binary search algorithm.
6. Recursive branching	f. A problem-solving technique where the same work is applied to an ever-subdivided portion of a task that resembles the overall total task at hand.
7. Binary search	g. When two recursive invocation chains emerge from one method invocation.
8. QuickSort	h. Algorithm that implements a "divide-and-conquer" technique to finding a value.
9. Pivot value	i. Logic that causes a recursive invocation chain to stop and begins the completion of recursed method execution.
10. Midpoint	j. The memory "cost" of solving a programming problem using a recursion technique.

Think about It

1. Describe the concept of recursion in the simplest terms.
2. What aspects of a programming language like Java enable the use of recursive approaches to problems?
3. For what types of problems can recursion be best applied?
4. Can all iteratively solved programming problems be converted to a recursive approach? When might this be suboptimal?
5. Describe what the base case does in a recursive method. Why is it important?
6. Describe the recursive method invocation chain. How does this limit some recursive approaches to programmatic problem-solving?
7. Describe recursive branching. How can it be useful when processing a series of data?
8. Why can a recursive method invocation chain be described in terms of last in, first out (LIFO)?
9. How does recursion enable a binary search's speed and effectiveness?
10. Describe the divide-and-conquer nature of the QuickSort algorithm.
11. How does a recursive approach lend itself well to the calculation of a Fibonacci number?

Short Syntax Problems

1. Write a short recursive method that will print all the letters from `a` up until a letter provided as a parameter in alphabetical order.
2. Write a short recursive method that will take a `String phrase` as a parameter and print all the vowels in the `phrase` in the reverse order that they occur.
3. Write a short recursive method that will search a `String` and print out the quantity of `e` and `i` vowels found.
4. Write a short recursive method that will print out lottery numbers randomly. Seven numbers should print, each number randomly chosen between `1` and `75`. The eighth number should be a `SuperNumber` and randomly be chosen between `1` and `150`. Ensure that the `SuperNumber` prints last.
5. Write a short recursive method that will accept an int parameter and print out a number of lowercase letters, randomly chosen, between `a` and `z`, inclusive.

Full Problems

1. Write a program that will accept a String phrase entered by the user on the keyboard. The program will perform the following tasks recursively:
   a. It will count the number of words in the phrase and display that count.
   b. It will process the number of vowels **in each whole word separately**. It will display that count at the end.
2. Write a program that will recursively search a file directory and list only the files that end with the `.txt` file extension. It will present this list to a user along with option numbers. If the user chooses a particular file, the program will open the file and print its contents to the console. The user will be presented with the option to choose another file or quit the application.
3. Write a program that implements a recursive method called `binarySum()`. The method will accept an array of int values and will recursively sum up all pairs of numbers in the array in a binary fashion (recursively working on the left and right ranges of the array in a divide-and-conquer fashion). Include a base case for a single value.
4. Leverage your chapter 8 skills and create a class called `LinePerson`. This class will have one data field, a double `personHeight`. Create an instance method for the class that will compare two `LinePerson` object heights (the instance object and the reference passed into the method). Then write an application that implements the QuickSort algorithm from this chapter. Adapt the algorithm so that an array of `LinePerson` objects are sorted by their `personHeight`, from smallest to largest.
5. Write a program that will prompt a user for two pieces of information: a file directory they would like to recursively search and a file extension they would like to search for. The program will recursively search the directory and list all files that match that file extension. If no files are found, print an appropriate message to the user.