An Introduction to Computer Science with Java

Contents Chapter 16 – Recursion

Introduction to Computer Science with Java

Chapter 16 – Recursion

Community College of Philadelphia Computer Science

This chapter discusses the topic of recursion. Recursion can be a powerful tool in computer programming or a colossal waste of resources, depending on how it is used. The chapter includes sections on mathematical concepts of recursion, recursion in Fractal geometry, and a discussion about when to use recursion in computer programming.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of recursion and related concepts, including the notion of a *mise en abyme* work of art or literature, infinite recursion, and algorithmic recursion.
- describe mathematical concepts related to recursive functions, including primitive recursion, total function, total recursive function, partial recursive function, and mutual recursion.
- describe how recursion is related to fractal geometry and give several examples of recursive fractal structures, such as the Cantor set, the Sierpinski triangle, and the Menger sponge.
- describe what is meant by a stack frame and stack space, and how this is important in considering when and how to use recursion in computer programming.
- describe the importance of a good base case in recursive algorithms and what happens without a good base case.
- describe the difference between depth-first and breadth-first processing of a dataset being broken into parts and how this is related to efficient recursion.
- create methods in Java that exhibit efficient use of recursion.

16.1 What is recursion?

In general, **recursion** occurs when an item is has one or more parts that emulate the overall item. By its very nature, recursion repeats itself. If the structure of the recursive part is an emulation of the overall item, then it will contain a copy of itself, that copy will contain a copy, and so on. If this continues infinitely, then we have **infinite recursion**. More practically, the recursion ends at some fundamental level, such the level of pixels in a digital image.

An example from the art world can illustrate this. The French phrase **mise en abyme**, which means "*placed in the abyss*", refers to an art technique in which an image contains a smaller copy of itself, which will contain a smaller copy of itself, etc., "into the abyss".

You can try this yourself with a mirror and a modern cell phone. The image on the right shows the author creating such a *mise en abyme* image with a Microsoft RT tablet.

This recursion effect is not limited to the world of images. The Nobel prize winning French author *André Gide* coined the term *mise en abyme* in the 1890's to refer to a work within a work, such as a play within a play, a book within a book, or a picture within a picture.

Algorithms can be recursive in the same way. Algorithms are composed of individual modules containing parts of the algorithm. An algorithm can be recursive if one or more of those parts are the same as the overall algorithm. This process, called **Algorithmic recursion,** is the process of an algorithm invoking a copy of itself as a module or part of itself. That part will, by definition, invoke a copy of itself, which will invoke a copy of itself, and so on.

In object-oriented programming, such as in the Java language, a **recursive method** is a method that calls itself.

Algorithmic recursion can continue indefinitely, or be stopped when some predefined base case is reached in the algorithm. Here is an example:

The factorial of a positive integer n, written as *n!*, is the product of all non-negative integers up to and including *n*.

```
4! = 1 * 2 * 3 * 4 = 24

8! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 = 40,320
```

Alternatively, we can think of *n!* defined downwards, as the product of all integers from *n* down to *1*.

```
4! = 4 * 3 * 2 * 1 = 24

8! = 8 * 7 * 6 * 5 * 4 *3 * 2 *1 = 40,320
```

Iteration can be used in an algorithm to calculate the factorial of a number. Remember that **iteration** means looping in a computer program, usually increasing or decreasing by 1 each time through the loop. Here is an iterative algorithm to calculate factorial coded as a Java method.

```
// a method to calculate n factorial iteratively upward
// works with positive integers up to the limit of the int data type
// 12! is the limit for int data
int factorial(int n) {

     int factorial = 1;

     for (int i = 2; i <= n; i++)
         factorial = factorial * i;

     return factorial;
 } // end factorial()
```

Here as another version of the algorithm calculating factorial, but this time downward, from *n* to 1 rather than from 1 to *n*.

```
// a method to calculate n factorial iteratively downward
// works with positive integers up to the limit of the int data type
// 12! is the limit for int data

int factorial(int n) {

     int factorial = (n);

     for (int i = (n-1); i > 0 ; i--)
         factorial = factorial * i;

     return factorial;
 } // end factorial()
```

Factorials can also be calculated recursively. Notice that *8!* contains *7!*

```
7! = 7 * 6 * 5 * $ * 3 * 2 * 1
8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
8! = 8 * (7 * 6 * 5 * 4 * 3 * 2 * 1)
8! = 8 * 7!
```

In general, *n! = n * (n-1)!* for all integers > *1*; and *1! = 1*

We can use this knowledge to create a recursive algorithm for calculating the factorial of a positive integer, which will stop at a base case of 1:

```
    // a method to calculate n factorial recursively
   // works with positive integers up to the limit of the int data type
   // 12! is the limit for int data

    static int factorial (int n) {

        if (n == 1)
            return 1;
        else
            return n * factorial(n-1);

    } // end factorial()
```

In this recursive method, the algorithm calls itself until it reaches the base case of n = 1. This is a typical pattern for recursive algorithms: repeat the recursion until a base case is reached.

Of course, the ability of any method to calculate factorials is limited by the range of values for the data type being used, just as it would be for a method to calculate X^2 or X^Y.

In the java code above, once we get an answer beyond the range of *int* values, our answer is invalid. The *int* data type has a minimum value of *-2,147,483,648* and a maximum value of *2,147,483,647* (roughly ± 2.1 x 10^9). The value of *12!* is *479,001,600* (roughly 4,79 x 10^8), while the value of *13!* is *6,227,020,800* (roughly 6.23 x 10^9) 13! is out of range for the *int* data type. Using *int* variables, we can only calculate factorials up to *12!*.

Java long integers have a range of accuracy that is roughly *± 9.2 x 10^{18}*. *36!* is within this range, but *37! is not.* Using long int values, we can only calculate factorials up to *36!.* Thisis true for both iterative and recursive methods that calculate factorials.

Before we look at issues related to recursion in programming, let's look at some formal mathematical concepts related to recursion, and at examples of recursion in the world around us.

16.2 Mathematical Concepts of Recursion

Every function has a domain and a co-domain. The **domain** of a function is the set of possible values for the argument of the function. The **codomain** of a function is the set of possible results that can be derived from the domain. If y is a function of x, written as *y = f(x)*, the set of possible x values is the *domain,* and the set of possible *y* values that result from those *x* values is the *codomain*. Usually the domain of a function is specified as part of the definition of a function. For example, consider the recursive function:

f(n): {if n = 0 then f(0)= 0, otherwise f(n) = f(n-1)+1},
where the domain is defined as all non-negative integers.

What is the codomain of this function? To figure out the answer, let's look at an example, *f(2)*:

$$
\begin{aligned}
f(2) &= f(1) + 1 && \text{// if n = 1, then } f(1) = f(1-1)+1 \text{ since } f(n) = f(n-1)+1\\
&= \overbrace{(\underbrace{f(0)}\ + 1)} + 1 && \text{// f(0)for this function is defined to be 0}\\
&= (\ 0 \quad + 1) + 1\\
&= 2
\end{aligned}
$$

Although this is not a proof, it shows the function can be reduced to *f(n) = n*. The codomain for this function is the same as the domain – the set of all non-negative integers.

For the function *f(n) = n/2*, with the domain of all integers, the codomain is not the set of all integers, but in the set of rational numbers, since *1/2,* for example, is a rational number, but not an integer.

The formal mathematical definitions of recursion are limited to cases where the domain and codomain of the function are both in the set of natural numbers as defined by the Peano axioms[1]. The Peano axioms use the successor function *S(n) = n + 1* to define the natural numbers. According to the *axioms*, the constant *0* is a natural number, and If *n* is natural number then its successor, *n+1,* is a natural number, which gives us the *set {0, 1, 2, 3, …}*. This is the set of non-negative integers.

So, given the constraints of working within the set of non-negative integers, there are several formal mathematical definitions related to recursion.

Primitive recursion is a procedure that defines the value of a function at an argument *n* by using its value at the previous argument *(n – 1)*. `f(n) = g( f(n-1) )`, where *n* is limited to the non-negative integers. In other words, primitive recursion occurs when the function's value at *n* is a function of its value at *(n-1)*. However, primitive recursion must have a base case, most commonly at *n= 0*. The recursive factorial function shown above is a primitive recursive function. It could be defined as:

> `f(n): {if n = 1 then f(1)= 1, otherwise f(n) = n * f(n-1) }`
> where the domain is defined as all non-negative integers.

Any function that is equivalent to a primitive recursive function, or that can be defined in terms of primitive recursive functions, is also a primitive recursive function, no matter how complicated its definition is. All primitive recursive functions can be implemented iteratively on a computer using count controlled loops, such as a *for* loop, without the use of more general *while* loops or *goto* statements.

a **total function** is a function whose definition determines an acceptable value of *f(n)* for every possible value of *n*. A **total recursive function** is a recursive function that is a total function.

A **partial recursive function** is a recursive function that does not yield an acceptable value of *f(n)* for every possible value of *n*. Remember, the set of non-negative integers is our domain and codomain in these mathematical definitions. Consider the function *f(n) = f(n-2)+1*, with a base case *f(0) = 0*.

> `f(n): {if n = 0 then f(0)= 0, otherwise f(n) = f(n-2)+1},`
> where the domain is defined as the set of non-negative integers.

For *f(4)*, for example, this function yields:

$$\begin{aligned} f(4) &= f(2) + 1 \\ &= \overbrace{(\underbrace{f(0)} + 1)} + 1 \\ &= (\ 0 \ + 1) + 1 \\ &= 2 \end{aligned}$$

[1] The Peano axioms are asset of axioms that from the basis for formal definitions of mathematics.

But what is the value of f(3)?

$$\begin{aligned} f(3) &= f(1) + 1 \\ &= \overbrace{(f(-1) + 1)} + 1 \end{aligned}$$

Since *-1* is not in the set of natural numbers, *f(-1)* is not defined as part of this function. The function is not a total function, so it is not an example of a *total recursive function*, but of a *partial recursive function* because it is undefined for some of its possible arguments.

All primitive recursive functions are total recursive functions, but not all total recursive functions can be reduced to primitive recursive functions. *Ackerman's function* is an example of a total recursive function that is not a primitive recursive function. (For more about Ackerman's function see the NIST Web page discussing the function, online at: http://xlinux.nist.gov/dads/HTML/ackermann.html .)

The term *general recursive function* is not clearly defined; some mathematicians use it to mean *partial recursive function*, others use it to mean *total recursive function*.

Mutual recursion is when two functions are defined recursively in terms of one another. Consider the following simple example in Java, which has been used for many years in other programming languages:

```
boolean isEven(int n)
{
    if(n == 0)
        return true;
    else
        return isOdd(Math.abs(number)-1)
}

boolean isOdd(int n)
{
    if(n == 0)
        return false;
    else
        return isEven(Math.abs(number)-1);}
```

These two functions alternate back and forth between one another, each time decreasing the passed argument by 1, until the base case is reached. If the base case is reached by *isEven()*, the original argument is even; if it is reached by *isOdd()*, the original argument is odd.

Mutual recursion is used in functional programming meta-languages such as ML and CAML, and in compiler code that parses mathematical expressions and language syntax.

To recap, a *primitive recursive function* is a function in which *f(n)* is a function of *f(n-1)*. A *total recursive function* yields an acceptable value for all possible arguments. A *partial recursive function* is a recursive function that does not yield an acceptable value for all possible arguments. All primitive recursive functions are total recursive functions, but not all total recursive functions are primitive recursive functions. The term *general recursive function* is not well defined. *Mutual recursion* occurs when two recursive functions are defined in terms of one another.

(*For more information on mathematical concepts of recursion, see Chapter 16, Section 16.2 recursion*.)

16.3 Recursion in Fractal Geometry

The Cantor Set

The Cantor set was described in 1883 by the German mathematician *Georg Cantor*, one of the founders of modern set theory. It is a remarkably simple idea with surprising consequences. It is a part of the foundation of fractal geometry. We will not dwell on the properties of the Cantor set, but simply show what it is.

Fractal geometry is a geometry of objects with a self-similar structure. It relies heavily upon recursive algorithms to create such structures.

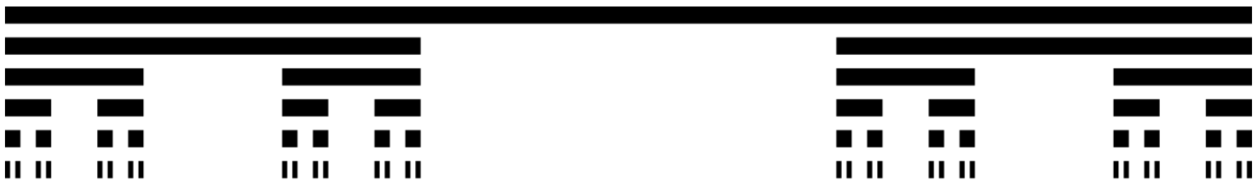
Algorithm to construct a Cantor set:

```
start with a line segment
cantor(line segment)
{
   divide the line segment into thirds

   remove the middle third

   cantor(first third)
   cantor(last third)
}
```

Here is an image of the process described above, repeated through five levels of recursion.

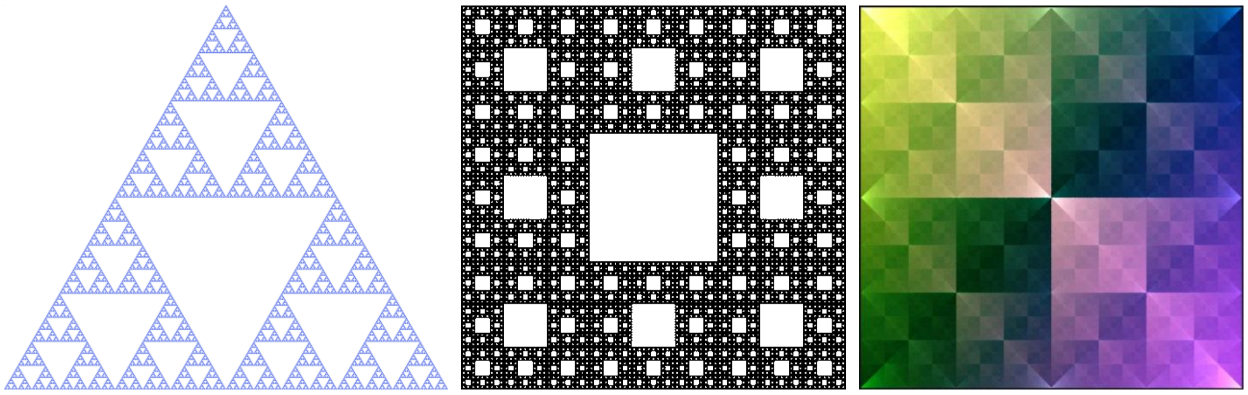


Each of the outer thirds of the Cantor set is self-similar to the overall Cantor set.

Sierpinski Triangle

The Polish mathematician Wacław Sierpiński [2]described several fractal images that can be created with recursive algorithms, including the Sierpiński triangle, the Sierpiński carpet, the Sierpiński square and the Sierpiński curve. Shown below are the Sierpiński triangle, the Sierpiński carpet, and the Sierpiński square.

[2] Sierpinski contributed to set theory and number theory. For more about Sierpinski see a biography online at: http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Sierpinski.html



Each of these is a self-similar fractal image that can be generated with a simple recursive algorithm. The term self-similar means that part of the structure of the object has the same structure as the overall object itself. In other words, it is a recursive structure.

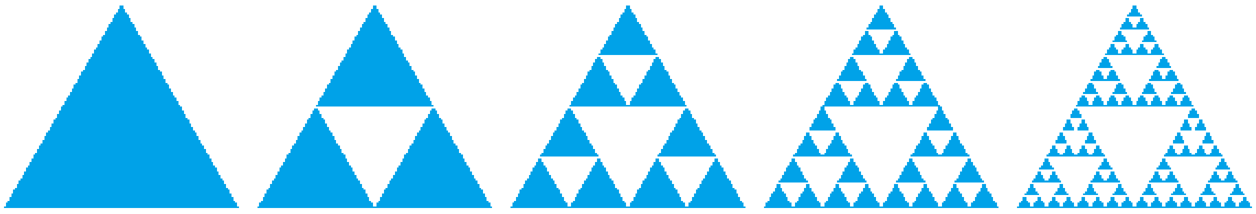
here is an algorithm for creating a Sierpiński triangle, also called a Sierpiński gasket or Sierpinski sieve:

```
// Algorithm to create a Sierpiński triangle

draw an equilateral triangle

sierpinski(triangle)
{
   find the midpoint of each side of the triangle
    draw lines connecting the midpoints.
        this creates four smaller triangles a, b, c, and d.
        d is the center triangle
    color in (or cut out) triangle d (the center triangle)
    sierpinski (triangle a)
    sierpinski (triangle b)
    sierpinski (triangle c)
}  // end sierpinski()
```

The Sierpinski triangle algorithm splits the triangle into four smaller triangles, and then calls itself for three of the four smaller triangles. Here is the result through five levels of recursion:



The process of dividing the triangle into smaller triangles and removing the center is repeated, each time at a smaller level, each time in more detail. The result is a complex structure created by a small, efficient algorithm.

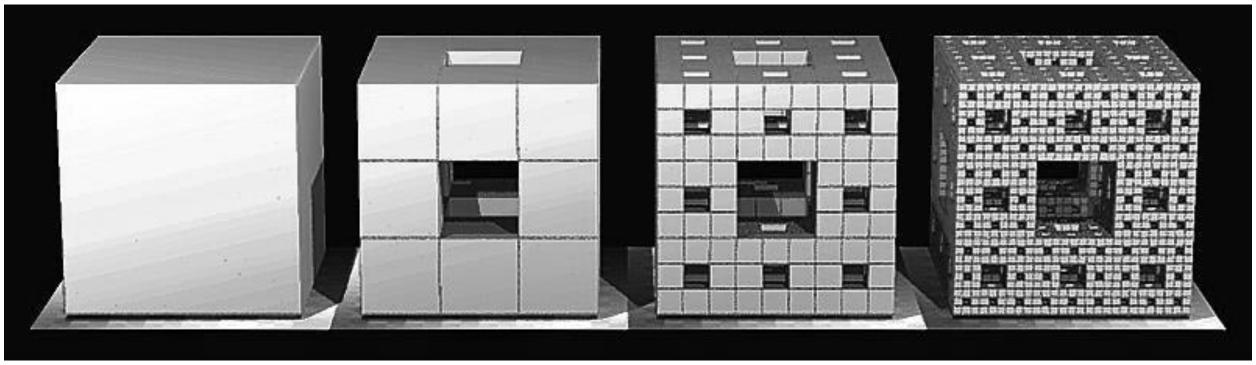
Images such as the Cantor set and the Sierpinski trangle are interesting, but do they have any applications in the real world? To answer that question, let's look at a three dimensional object based on the Sierpinski carpet, the *Menger sponge*.

Menger sponge

Imagine that we start with a cube, then divide each face of the cube into 9 squares, much like a tic-tac-toe board. Then, we cut the center square from the cube all the way through to the other side. Repeat this for each face of the cube. We actually only need to do this three times, since each cutting goes through from one face of the cube to the opposite face.

The original lines we drew on the cube can be thought of as dividing the cube into 27 smaller cubes.

Repeat this process for each of the reaming smaller cubes, and so on. The diagram below illustrates this process through three levels of recursion.



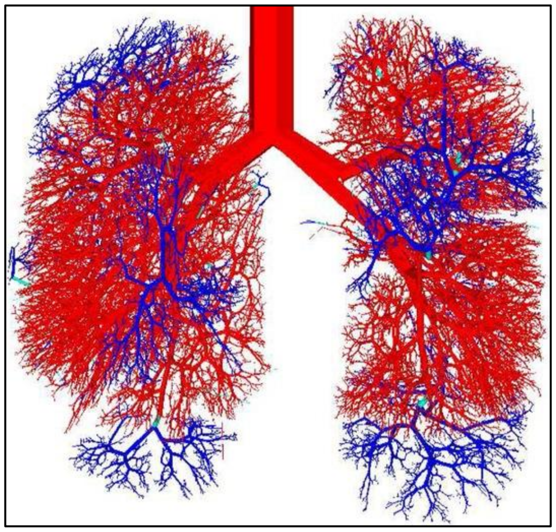
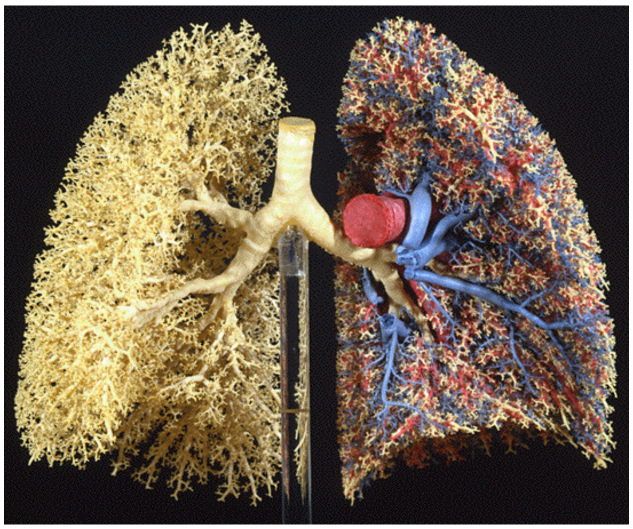
Each time we cut through the cube, we are removing some of the mass of the cube. From the solid cube to the first level, we are removing 7/27 of the mass of the cube.

At the same time, we are increasing the surface area of the object. From the solid cube to the first level, we are removing six surface squares, but adding the surface are of the “tunnels” into the cube, which add up to 24 squares, for a net of 18 additional squares of surface area. Before the first cut, the square had six sides with 9 squares on each side for a total surface area of 54 squares. With the cut, we lost 6 surface squares but gained 24 tunnel squares, for a total surface area after the cut of 72 squares.

The surface area of the object is increasing at the same time that its mass is decreasing. This makes the Menger sponge an interesting object of study for learning about structures such as lungs, circulatory systems in animals, and root and leaf systems in plants where the efficiency of a structure with maximum surface area for a given mass is important. There are many fractal structures created by recursive algorithms that exhibit such interesting properties.

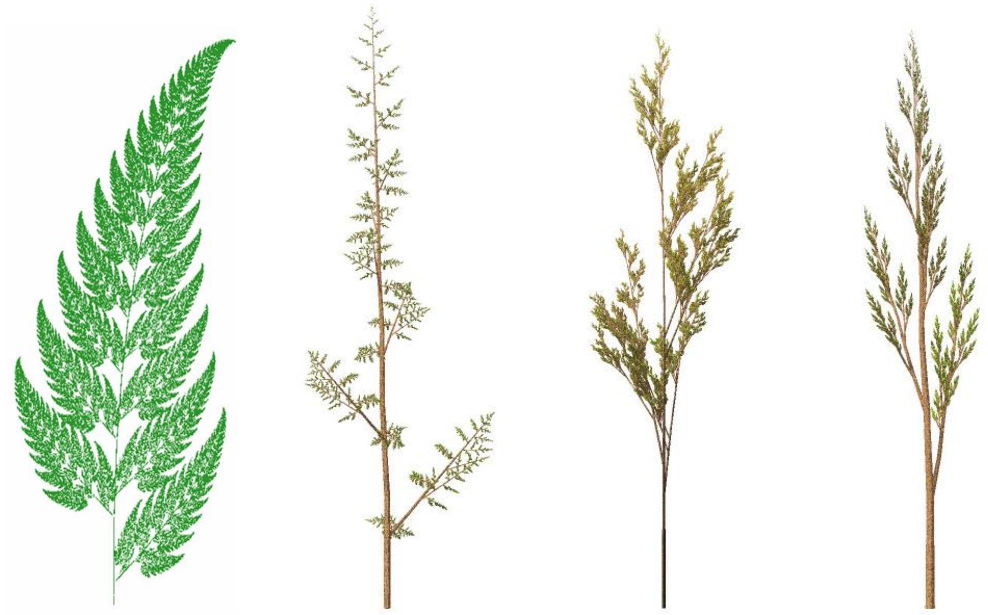
In the human cardio-pulmonary system, oxygen passes from very small alveoli in the lungs to red blood cells in very small capillaries in the body. The ideal structure for both the lungs and the blood vessels would be to have maximum surface area with a minimum of mass, just like the Menger sponge.

The images below are, on the left, a three-dimensional model of a lung-like structure created with a simple recursive algorithm, and, on the right, a resin cast of the vessels in human lungs.


.


It turns out that structures like the Menger sponge are good models for understanding the formation and function of many biological systems. Fractal geometry and recursion are fundamentally important for understanding many systems in the natural world. From seashells to solar systems, from grapheme to galaxies, the universe around us is filled with examples of structures and processes that can be understood recursively.

Consider the four images below. Can you tell which of these are real plants and which are computer generated images made by recursive algorithms?



All four of these images were created on a computer using recursive algorithms. For more about fractals and recursion in the natural world, see the following:

The Algorithmic Beauty of Plants;
 by Aristid Lindenmayer, et. al.; Springer, 1991; ISBN: 978-0387972978

The Computational Beauty of Nature;
by Gary Willima Flake; Bradford Books, 2000; ISBN: 978-0262561273

Other important topics related to recursion in fractal geometry include Julia sets and the Mandelbrot set. The Mandelbrot program included with the files for Chapter 13 and this chapter is a recursive algorithm that draws an image of the Mandelbrot set.

16.4 When to Use Recursion in Computer Programming

Recursion in computer programming can yield elegantly efficient results, or can be an unnecessary and colossal waste of resources. To see why this true, and to learn how to decide when to use recursion, let's start by revisiting the recursive factorial program we saw earlier in the chapter.

```
static int factorial (int n) {

    if (n == 1)
        return 1;
    else
        return n * factorial(n-1);

} // end factorial()
```

The factorial program is an example of primitive recursion -- the value of factorial for argument *n* is defined by using its value at the previous argument, *(n – 1)*.

It is also an example of tail recursion. **Tail recursion** occurs when the recursive call is the last instruction in the method that is being called recursively.

In addition, recursive factorial program is an example of **linear recursion**, which means that each copy of the program calls itself only once.

All three of these factors – primitive recursion, tail recursion and linear recursion are indications that it shouldn't be too hard to rewrite the process as an iterative process using a loop instead of recursion. As we saw, the iterative version closest to the recursive algorithm is:

```
int factorial(int n) {

     int factorial = (n);

     for (int i = (n-1); i > 0 ; i--)
         factorial = factorial * i;

     return factorial;
 } // end factorial()
```

So, which of these should we use, and why? When both an iterative method and a recursive method similar to one another are available, it is almost always best to use the iterative version because of recursion overhead, described below.

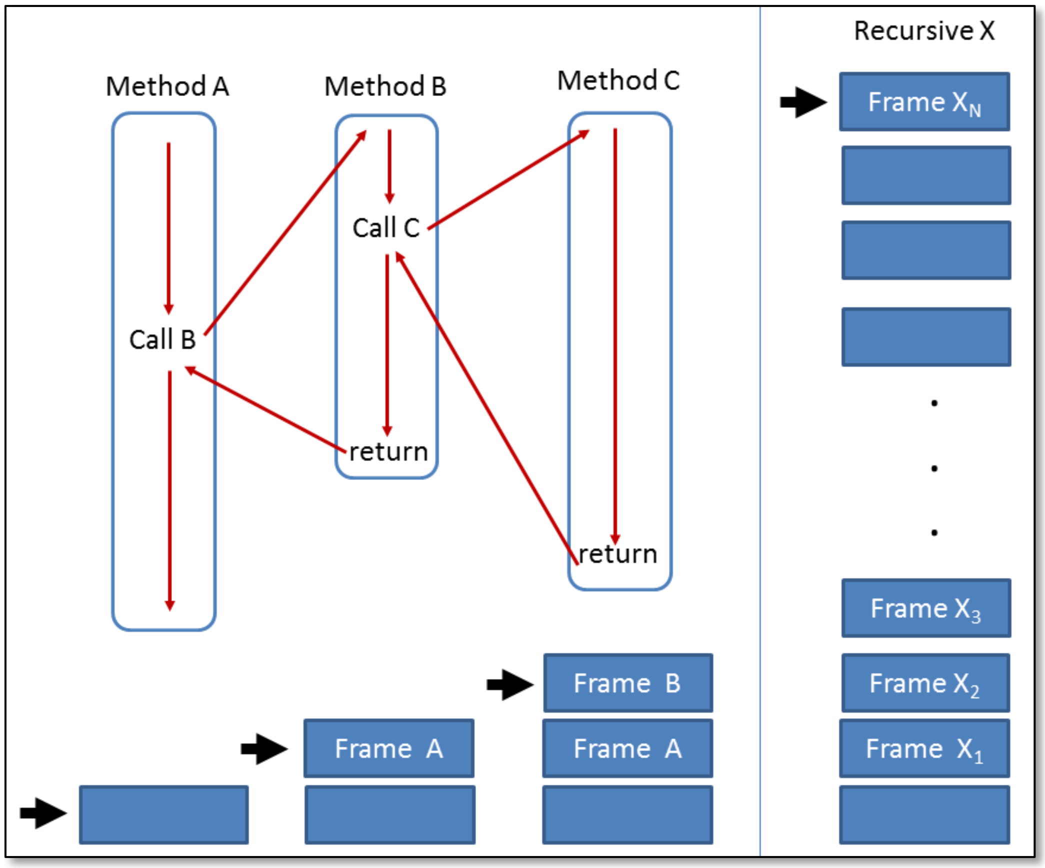
Recursion Overhead

Recursion overhead is the additional memory and processing time used by recursive calls to the same method.

Each time a method calls another method, information about the change in methods needs to be stored on the computer's system stack. The example below illustrates how this works.

Imagine that *methodA* is running. In the middle of *methodA*, there is a call to *methodB*, so that control is transferred to *methodB*. The computer must store information about where it was in methodA when this call to methodB occurred, the status of the variables in *methodA* at the time, and so on. It stores this information on a Last-In-First-Out (LIFO)data structure called a *stack*. It creates a set of data about the status of *methodA* called a *stack frame*, and puts the stack frame on the system stack. It then starts *methodB*.

Now imagine that *methodB* calls *methodC*. Once again the computer creates a stack frame with all the necessary information about where it was in *methodB*, etc., puts this onto the system stack, then starts *methodC*. When *methodC* is done, the stack frame for *methodB* is retrieved and *methodB* continues from where it left off. Then when *methodB* is finished, the stack frame for *methodA* is retrieved and *method*A continues. The diagram below illustrates this. On the left, we see the flow of program control from method to method and the stack frames created by this. On the right, we see a large stack due to recursion. Stack space is the major part of the overhead caused by recursion.



The stack frames keep track of what happens when a method is finished running. They take up space in the memory of the computer. Usually, a certain amount of stack space is allocated before a program starts running. If there are too many calls to other methods, then the program can run out of stack space. A **stack overflow exception occurs** when a Java program runs out of stack space. This can happen with recursive algorithms.

When a method calls itself, it still needs a stack frame to keep track of the transfer of control from one copy of the method to another. This is the overhead caused by recursion. Iterative methods, which repeat something in a loop, don't need additional stack frames, they simply change the value of existing variables in memory.

The two programs – *iterativeCount* and *recursiveCount* illustrate this. They are included as NetBeans projects with the files for this chapter. You should try them both and compare how they run.

```
/* IterativeCount.Java
 * Computer Science 112, Spring 2014
 * Last edited Jan. 20, 2014 by C. Herbert
 *
 * This code is a simple iterative program for printing the integers from
 * 1 to the highest integer, Integer.MAX_VALUE
 *
 * It is intended as a comparison to the recursive program
 * RecursiveCount, which does the same thing recursively.
 *
 * You should run the two programs and compare results.
 * The iterative program will run on, until you stop it or until it reaches
 * Integer.MAX_VALUE.  It speed is determined by the time it takes to execut the
 * System.out.println() instruction.
 * Without the printing, it get to Integer.MAX_VALUE in less than 2 seconds
 * on most systems, often in less than 1 second
 */
package iterativecount;

public class IterativeCount {

    public static void main(String[] args) {
       int count;   // a counter

       // Integer.MAX_VALUE is the highest possible int value
       for(count = 1; count < Integer.MAX_VALUE; count++ )
            System.out.println(count);

} // end main()

} // end class IterativeCount
```

```
/* RecursiveCount.Java
 * Computer Science 112, Spring 2014
 * Last edited Jan. 20, 2014 by C. Herbert
 *
 * This code is a simple recursive program for printing the integers from
 * 1 to the highest integer, Integer.MAX_VALUE
 *
 * It is intended as a comparison to the iterative program
 * IterativeCount, which does the same thing iteratively.
 *
 * You should run the two programs and compare results.
 * This recursive program will crash and generate
 *    'Exception in thread "main" java.lang.StackOverflowError'
 * followed by many other errors as it backs out of the recursion.
 *
 * Scroll up through the output to find the first error message.
 *
 * This illustrates the problem with recursion overhead, and shows that
 * recursive techniques such as merge sort and quick sort are executing a
 * "depth first" recursion, otherwise they  would run out of stack space
 * with large lists, but they do not.
 *
 */
package recursivecount;

public class RecursiveCount {

   public static void main(String[] args) {

      // start the recursion
      reCount(1);

   }  // end main()

   public static void reCount(int count) {
       System.out.println(count);

      // Integer.MAX_VALUE is the highest possible int value
       if (count < Integer.MAX_VALUE)
           reCount(count + 1);

   } // end (recount()
}  // end class RecursiveCount
```

So, if recursion has such overhead, then why should we ever use recursion?

There are several cases in which recursive algorithms can solve problems far more efficiently than iterative algorithms. *Merge Sort*, *Quick Sort* and *Heap Sort* are three recursive sorting methods that far outperform iterative sorting methods such as *Bubble Sort*, *Selection Sort* and *Insertion Sort*. We shall look at sorting methods later this semester.

A binary search algorithm is effectively a recursive algorithm, not an iterative algorithm. Perhaps you remember a game something like this: "*I am thinking of a number from one to one hundred. Try to guess what it is. I will tell you if you are correct, high, or low.*" The strategy for finding the right number, on the average, in the fewest number of guesses, is a recursive strategy.

The most efficient algorithms for finding the shortest path from one point to another in a network of nodes is a recursive algorithm. The algorithms used by Web sites such as Google Maps and the algorithms used to route traffic through the Internet are recursive algorithms.

What do all of these efficient recursive algorithms have in common? Three things:

1. They are examples of non-overlapping "*divide and conquer*" strategies.
2. The recursion has a well-defined base case.
3. They execute their recursion in a "*depth-first*" manner, rather than a "*breadth-first*" manner.

We will look at what each of these things means in the next sections.

Divide and Conquer Recursion

Let's go back the guessing game mentioned above. The problem is to find a randomly selected number between 1 and 100 in the fewest number of guesses. The other player will tell us if our guess is correct, too high, or too low. (The other player may be a computer.)

What strategy can we use to do this? We could try a linear search:

Player A: Is the number 1?
  Player B: too low. Try again.

Player A: Is the number 2?
  Player B: too low. Try again.

Player A: Is the number 3?
  Player B: too low. Try again.

. . . and so on, until Player A finally guesses the right number.

A linear search like this is a correct solution to the problem, but unfortunately, it is terribly inefficient. It is the kind of solution a four-year old might try.

Here is a more efficient strategy, a binary search:

Player A: Is the number 50?
  Player B: too high. Try again.

Player A: Is the number 25?
  Player B: too low. Try again.

Player A: Is the number 37?
  Player B: too low. Try again.

. . . and so on until Player A guesses the right number.

Using this approach, Player A will always get the right number in seven guesses or less. Why? …because Player A is cutting the problem in half with each guess. No matter what the number is, there is only a one in 100 chance the player will get it right on the first guess, but by picking 50 – the midpoint of the range of numbers, the player's odds are increased to one out of 50 on the second guess, then one out of 25 on the third guess, and so on until the odds of getting the right number are one out of one on the seventh guess.

This is a classical divide and conquer strategy for problem solving – and it is recursive. The problem at the second level is a smaller version of the same problem as at the first level. The problem at the third level is a smaller version of the problem at the second level, then a smaller version at the fourth level, and so on, until we have a very small problem that is most easy to solve.

Let's look at the math. How many times can we divide 100 in half before we end up with a single item? If you know the math, then the answer is obvious: the log in base 2 of 100 ($\log_2(100)$) rounded up the next highest number. $\text{Log}_2(100)$ = 6.643856, rounded up = 7.

We don't need to know how to calculate logarithms to see that 7 guesses will work. Logarithms are the opposite of exponentiation. $\log_2(X)$ =? is basically asking "*to what power do we need to raise 2 to end up with X*"? It is the same as $2^? = X$. In the case of our guessing game, the question would be $2^? = 100$. However, we only need an integer answer, since we can't have 6.643856 guesses – we can have 6 guesses or 7 guesses, but not 6.643856 guesses.

If we start with 1 and multiply by 2, then do it again, and then again, and so on until we pass 100, we will multiply by 2 seven times. 100

↓

1 2 4 8 16 32 64 128

$2^6 = 64$. $2^7 = 128$. We can multiply by 2 seven times before reaching 100, and conversely, if we start with a set that has 100 items, we can only divide it in half (divide it by 2) at most seven times before we have just one item.

2^{10} is 1,024. Start our guessing game with numbers from 1 to a thousand, and we will always get the right answer in 10 guesses or less; 20 guesses for 1 million items; and 30 guesses for 1 billion items.

This illustrates the power of a recursive algorithm. If we can break a problem down into separate small parts similar to the overall problem, then we can write a recursive method to solve the problem quickly – very quickly.

```
// algorithm to guess a number;   n is the guess
// range is the size of the set of numbers,
// assuming we start with 1 to range, such as 1 to 100.

n = range/2;

guess (n, range)
{
if (n is not correct) then
    if (n is too low)
            guess ( (guess + range/2), range/2 );
```

```
   else      // if n is too high
          guess ( (guess - range/2), range/2 );
   } // end guess

// when guess ends, n will be the correct number.
```

Merge Sort is another good example. Basically merge sort works in two steps – we break a list down into many lists of length 1, then we put the list back together, sorting as we do so. We put lists of length 1 into sorted lists of at most length 2, then put the lists of length 2 into sorted lists of at most length 4, then into sorted lists of at most length 8, and so on, until we have our complete original list in order.

We will see the code for merge sort later in the semester. It is another classical example of a divide and conquer strategy. It is hard and time consuming to sort 1 million items, but it is easy to put two short lists that are already sorted into one bigger sorted list. If you look back at the Sierpinski triangle algorithm earlier in this chapter, you will see that it is also an example of a divide and conquer strategy.

We need to be careful with divide and conquer strategies – the separate parts into which we break the problem must not be overlapping. I f the parts overlap, then the recursion will be inefficient. To see an example of this, let's look at a recursive algorithm for calculating a Fibonacci number.

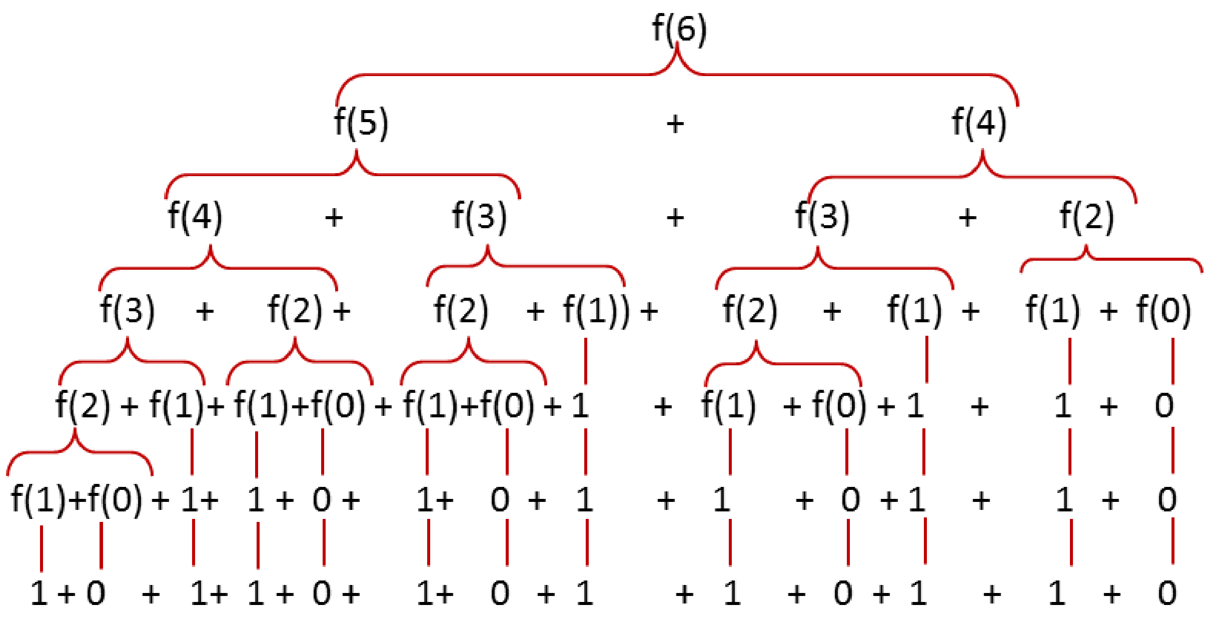
You may recall that each Fibonacci number is the sum of the two previous Fibonacci numbers. The set of Fibonacci numbers is {0, 1, 1, 2, 3, 5, 8, 13, 21, …}. There is an elegant recursive algorithm for calculating the nth Fibonacci number:

```
// algorithm for finding the nth Fibonacci number

fib(n)

\\\\\\\\\\\\\\\\\\\\\\\\\\    if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fib(n-1) + fib(n-2);
```

The problem is, this simple little algorithm is inefficient. The diagram below shows us why.


f(6)
f(5) + f(4)
f(4) + f(3) + f(3) + f(2)
f(3) + f(2) + f(2) + f(1)) + f(2) + f(1) + f(1) + f(0)
f(2) + f(1)+ f(1)+f(0) + f(1)+f(0) + 1 + f(1) + f(0) + 1 + 1 + 0
f(1)+f(0) + 1+ 1 + 0 + 1+ 0 + 1 + 1 + 0 + 1 + 1 + 0
1 + 0 + 1+ 1 + 0 + 1+ 0 + 1 + 1 + 0 + 1 + 1 + 0

Can you see that f(4) id calculated twice, f(3) is calculated 3 times, f(2) is calculated 5 times, f(1) is calculated 8 times and f(0) is calculated 5 times? That's a lot of work just to figure out that the sixth Fibonacci number is 8.

In this case, the elegant recursive solution, which seems obvious, is terribly inefficient. Why? … because the problem is broken down into overlapping parts. For a recursive solution to work, the parts must not overlap.

Think of what would happen in the following case. We wish to paint a large floor; so large that we will break it down into thirds. However, instead of hiring three people to each paint one third, we hire two people to each paint 2/3s. The floor will be painted, but 1/3 of the floor will be unnecessarily painted twice.

Watch for overlapping parts when using a divide and conquer strategy. The same recursion that multiples the efficiency of an algorithm can also multiply the inefficiency of an algorithm.

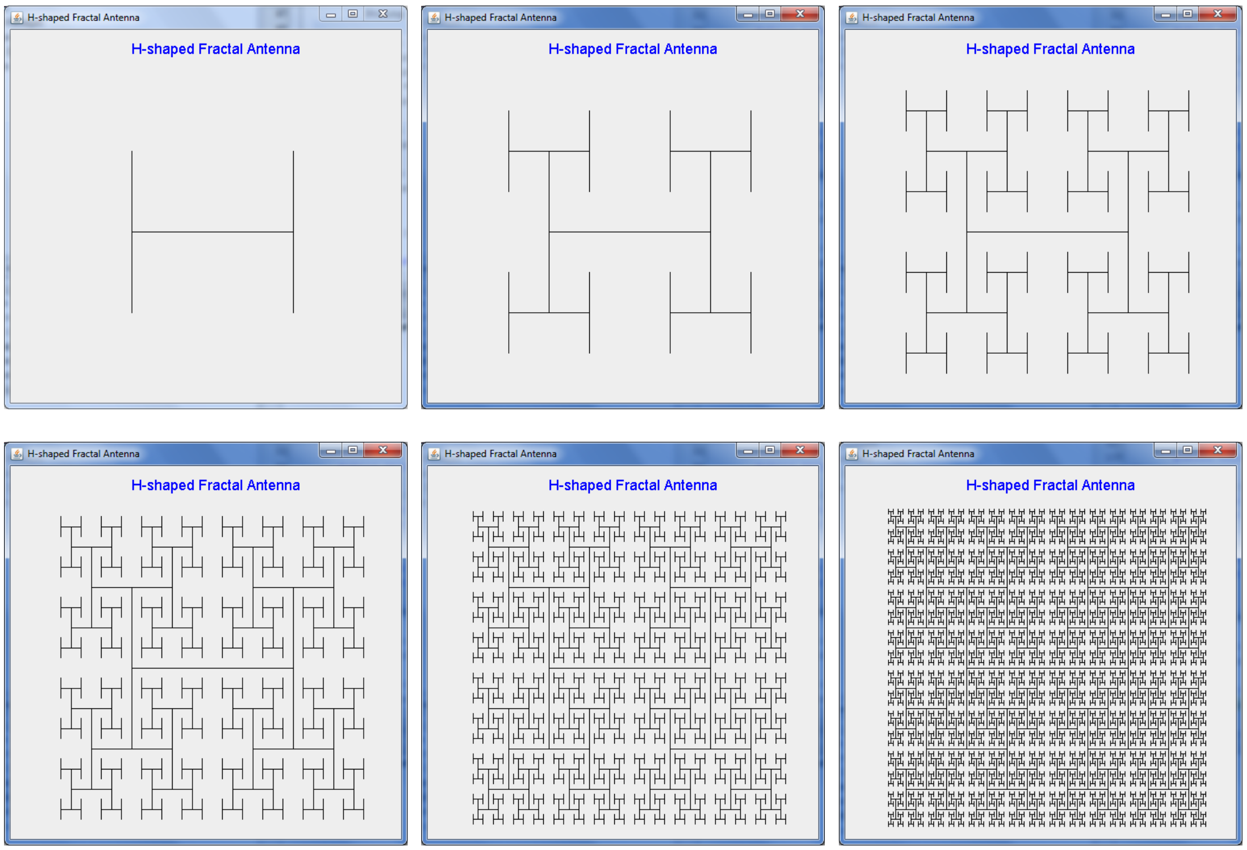
A Well-Defined Base Case

Are you familiar with proof by induction? First we prove that a proposition is true for 1, then we prove that if it is true for *n*, it must be true for *n+1*. Thus, we have proved that the proposition is true for all positive integers. The first step is called the base step. The second step is called the inductive step.

Recursion is a lot like this, but in reverse. We have a recursive case and a base case in a good recursive algorithm. Without the base case, the algorithm will continue indefinitely, or, more likely, if it is implemented as a computer program, it will continue until the program crashes.

Consider the following algorithm, which draws an H-shaped fractal antenna. Fractal antennas are the kind of antennas used with modern cell phones. They have only been around since the mid-1990s. Before that, antennas were made in particular lengths to work with particular radio frequencies. Nathan Cohen, from Boston University, discovered that the shape of the antenna was not as important as the length, and was experimenting with fitting long antennas into small spaces by folding them. He eventually discovered that fractal shapes could be used to make very long antennas fit into small spaces. A nice byproduct of this is that the fractal antennas can handle many different frequencies, not just a single frequency. Fractal antennas mean we don't need to have long antennas sticking out of our cell phones to improve reception.

One type of fractal antenna is an H-shaped antenna. to create one, we draw an H, then at each of the endpoints of the H, we draw another H, centered on the endpoint and half the size of the previous H. We repeat this process again and again until we have the desired antenna. The images below show this for six levels of recursion. They are the output from the program *DrawHdemo* included with the files for this chapter as aNetBeans project. The code uses mutual recursion, one method to draw the horizontal part of the H and one method to draw the vertical parts.



The algorithm continues until the lines that make up each H are only 5 units long. The base case for this algorithm is an H that is five units across and five units high.

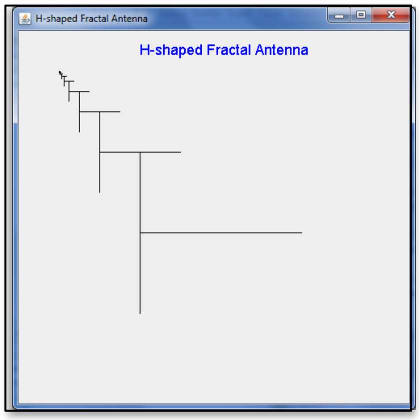
Here is a rough pseudocode description of the algorithm:

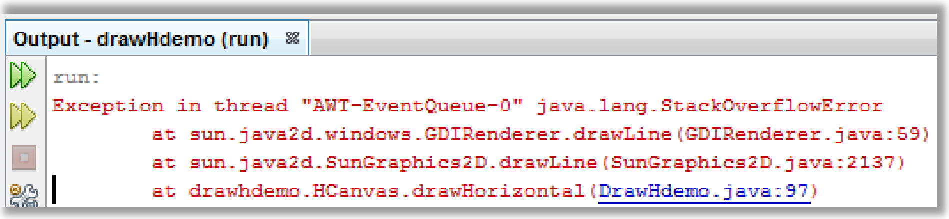
```
Set the mid-point for the first H to be in the middle of a square jFrame.
Set the length of the H to be (size of the JFrame)/2

drawH (midpoint, length)
{
   // the H will have four endpoints  end1, end2, end3, and end4
   if (length > 5)
   {
      length = length/2;
      drawH(end1, length)
      drawH(end2, length)
      drawH(end3, length)
      drawH(end4, length)
   }// end if
} // end drawH()
```

This is a good example of an elegant, efficient, recursive algorithm. It breaks a problem down into smaller parts that are similar to the overall problem, the parts don't overlap, and it has a base case.

Here is what happens to the same algorithm if we remove the base case:





```
Output - drawHdemo (run)
run:
Exception in thread "AWT-EventQueue-0" java.lang.StackOverflowError
        at sun.java2d.windows.GDIRenderer.drawLine(GDIRenderer.java:59)
        at sun.java2d.SunGraphics2D.drawLine(SunGraphics2D.java:2137)
        at drawhdemo.HCanvas.drawHorizontal(DrawHdemo.java:97)
```

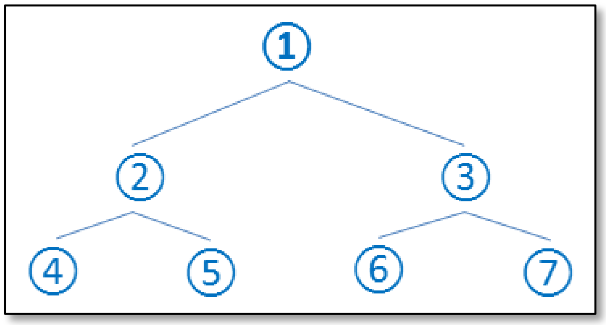
This output is from the same program, but with the *if* command to stop the recursion removed. It has no base case. What happens? it doesn't even draw one H. That is because it keeps trying to draw smaller and smaller parts of the antenna until there are so many levels of recursion we have a stack overflow.

This example illustrates the need for a good base case in the successful use of recursion in computer programming.

Depth-First Recursion

The example above also shows us that recursion is "depth first", not "breadth first".

A **depth-first** algorithm goes all the way down one branch of the algorithm then begins to back out, while a **breadth-first** algorithm will process the algorithm at one level completely before going down to the next level. Consider the binary tree shown here. It represents an algorithm that divides a data set in half, then in half again. Each of the numbered nodes represents a copy of the algorithm.



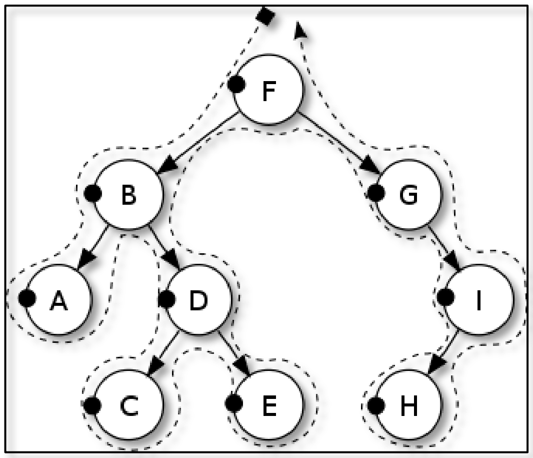
A breadth-first algorithm would process 1, then 2 and 3, then 4, 5, 6, and 7. It would complete one level before starting the next. A depth first search would process 1, then 2, then 4, then 5, then 3, then 6, and finally 7 .

We can see this in the case of the H program, above, that crashed. It did not finish one big H, then draw four little Hs, then sixteen smaller Hs, them 64 smaller Hs and so on. Instead it drew the crossbar of the H, then the left upright, then a smaller crossbar, then a smaller left upright, and so on. When the program crashed, it was attempting to draw the upper left part of (the upper left part of (the upper left part of (etc. of(the fractal antenna)))). This shows us that recursion is depth first. The computer follows one "branch" of the program all the way through to the base case, then backs out before starting another branch.

Why is this important? Let's go back to algorithms that deal with millions of items, such as the merge sort algorithm. The first part of merge sort breaks a list into two parts, then breaks those lists into two parts, then breaks those lists into two parts, and so on. Consider merge sort on a million items.

A "breadth-first" approach to the problem would be to break it into two parts, the break those into four parts, and so on. A separate copy of the method needs to be running for each part. If the computer processed all of the parts at one level before processing the next level, that means, in a list with a million items, merge sort would need to be running 1 million copies of itself at the lowest level of recursion. This is impossible on most computers. Programs don't have enough stack space. In fact, on an Intel *i-5* system with 8 Gb of RAM using the standard settings for *Java 7.3*, the recursive counting program shown earlier in this chapter could only handle less than 7,000 stack frames.

Instead of processing each level before going down to the next level, a recursive method will follow one "branch" of the algorithm all the way down to the bottom, then back up one level and go down the next path to the bottom, and so on. It will not do each level completely before going to the next level.



The diagram at the right shows this. Each circle is a copy of the method being called recursively. The algorithm starts with the top copy, copy F in the diagram. It then starts copy B for the left half of the data set. Then it starts copy A for the left half of B's data set. When copy A is done, it returns to copy B, then starts copy D for the right half of B's data set, then starts copy C. When C is done it returns to D, then starts copy E. When E is done, it returns to finish D. When D is done it returns to B. When B is done it returns to F, then starts copy G for the right half of Fs data set, then starts copy I, then starts copy H. when H is done it returns to I, then to G, then finally to F and the algorithm is complete.

Merge Sort, operates in a similar manner. The most number of copies of the method that are running at any one time is equivalent to the number of levels of recursion, which like our guessing game, above, is describe by logarithms and exponentiation – base two logs if the algorithm breaks into two copies of itself. With 1 million items, that means about 20 copies of the method running at most at one time. Well-constructed recursive methods should not run out of stack space.

Fortunately for us, the natural order of calls to methods within a recursive algorithm is depth-first rather than breadth-first. As long as we don't do anything strange to make our algorithms breadth-first, a divide and conquer method that calls itself should be able to process billions, even trillions of items without running out of stack space.

To summarize, recursion can be a powerful programming tool or it can be a colossal waste of resources. Recursion that uses a non-overlapping divide and conquer strategy, that has a well-defined base case, and that executes in a depth-first manner, rather than a breadth-first manner, is an efficient use of computing power, and the only way to solve many complex problems in computing, like some you will see later in this course and in a good data structures course.

Primitive recursion, linear recursion (especially linear tail recursion), recursion with overlapping parts, and recursion without a good base case should usually be avoided.

Chapter Review

Section 1 introduced the concept of recursion. Key terms included recursion, infinite recursion, *mise en abyme*, algorithmic recursion, and recursive method and iteration.

Section 2 discussed mathematical concepts of recursion. Key terms included domain, codomain,

Section 3 looked at several examples of recursion in fractal geometry., including the Cantor set, the Sierpinski triangle, the Menger sponge, and recursive structures in nature. The term fractal geometry was introduced.

Section 4 examined when to use recursion in computer programming. Key terms included: linear recursion, tail recursion, Recursion overhead, stack overflow, depth-first, and breadth-first. .

Chapter Questions

1. In general, when does recursion occur in the structure of an item?
2. When is an algorithm recursive?
3. What is meant by the term "base case" in recursion?
4. How are the domain and codomain of a function related to one another?
5. Formal mathematical definitions of recursion are limited to what domain and codomain?
6. What function do the Peano axioms use to define the natural numbers?

7. How is *f(n)* defined in primitive recursion?
8. What is thew difference between a total recursive function and a partial recursive function?
9. How are primitive recursive functions related to total recursive functions?
10. How does mutual recursion work?
11. Which class of recursive functions can always be implemented iteratively on a computer using count controlled loops?
12. Where is mutual recursion used in the world of computing?
13. What is true about *general recursion*?
14. How is fractal geometry related to recursion?
15. What properties of the Menger sponge make it an interesting object of study for learning about structures in the natural world?
16. When does linear recursion occur?
17. When both an iterative method and a recursive method similar to one another are available, which is better to use, and why?
18. What is the major part of recursion overhead?
19. What is a system stack used for?
20. Why is the recursive solution for finding Fibonacci numbers shown in the chapter not a good solution to the problem?
21. Why is a well-defined base case important in recursive algorithms?
22. When a large dataset is broken down into parts, and those parts are broken down into parts, and so on, what is the difference between depth-first and breadth-first processing of the data set?

Chapter Exercises

1. A Royal Story

Explain how the following *mise en abyme* story is recursive. What kind of recursion does it exhibit?

> *Once upon a time a young princess was sitting with her father, the king. She said to her father, "My Royal Father, can you read me a story?"*
>
> *"Yes my Royal Daughter" said the king, as he opened the Royal Storybook. He began to read, "Once upon a time a young princess was sitting with her father, the king. She said to her father, 'My Royal Father, can you read me a story?' . . .*

2. Top Secret Recursion

Assume that a computer student wants to impress his father, who is a computer security expert for the CIA, by writing a program like the one below. The parameter for the program is the network address of a computer system.

```
searchForDadsAccount(computer system)
{
   if (dad has an account on this computer)
      Print "Hello dad, I found your account."
   else
      find two other computers connected to this one
      searchForDadsAccount(first computer)
      searchForDadsAccount (second computer)
} // end searchForDadsAccount()
```

Even this code is enough to cause some problems, but the student made a mistake, leaving out the if …else instruction to check for the base case, so that the algorithm looks like this:

```
searchForDadsAccount(computer system)
{
      Print "Hello dad, I found your account."
      find two other computers connected to this one
      searchForDadsAccount(first computer)
      searchForDadsAccount (second computer)
} // end searchForDadsAccount()
```

A. What kind of recursion is within this second program?

B. Assuming that the student knows enough systems programming to make the code work, what will the program actually do?

C. If it takes 10 seconds for the algorithm to run, including the time it needs to connect to other computers, how many copies of the program will be running 1 minute after the program first starts? After 2 minutes? After 1 hour? After 6 hours?

D. What would something like this do to the Internet?

E. What sentence did US District Court Judge Howard G. Munson give Cornell University graduate student Robert Tappan Morris when he was convicted of trying to do something similar to this in November, 1988?

3. The program *drawHdemo* included with the files for this Chapter uses mutual recursion to draw a fractal H-antenna. Modify the program so that it has just one recursive method instead of two mutually recursive methods.

4. The Greatest Common Denominator GCD of two positive integers is the largest integer that divides both of them evenly (with a remainder of zero). A recursive version of Euclid's algorithm for finding GCDs is as follows:

```
gcd(int a, int b)
{
    if (b = 0)
       return a;
    else
       return gcd( b,(a mod b) );
    // mod is the integer remainder function
} end // gcd()
```

An iterative version is:

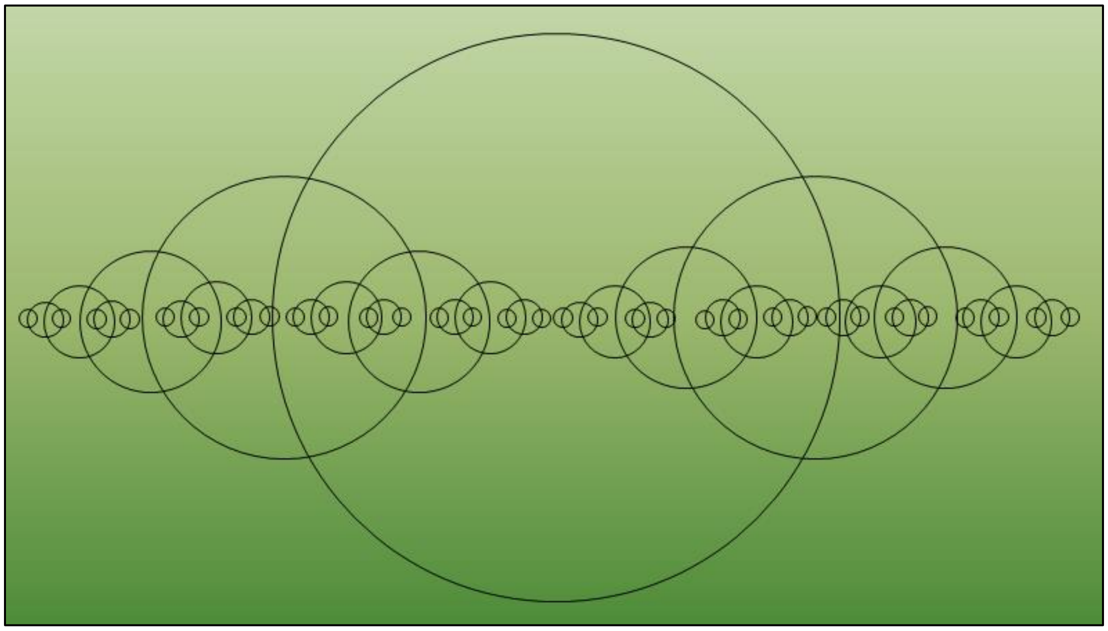
```
function gcd(int a, int b)
{
    while (a ≠ b)
    {
       if (a > b)
            a = a − b
       else
            b = b − a

       return a
   } //end while

} // end gcd()
```

Write a program that has both methods for finding a gcd and it with large random integers. Which of th two methods is more efficient, and why?

5. The image below shows a fractal pattern composed of circles along a line repeated through six levels of recursion.



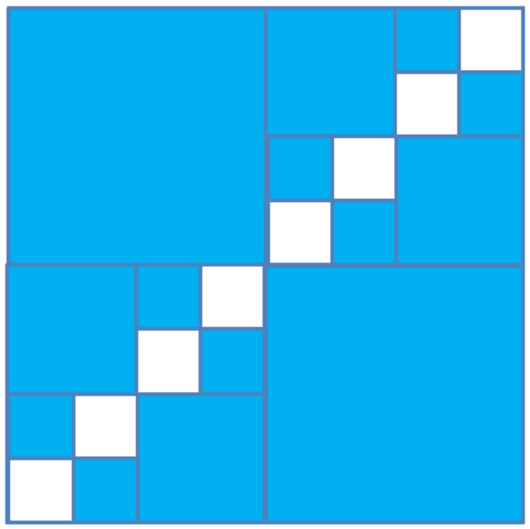
It draws a circle with a radius of *n*, and a center point on an invisible line. It then recursively draws two circles with a radius of n/2 and center points on line where the circles cross the line.

Write a Java program to do this through 10 levels of recursion.

6. Here is an algorithm for creating a fractal image:

```
Start with a square
fractal(square)
   {
   divide the square into four smaller squares
   // do this by drawing four squares, each with a side
   // half the length of the big square, lined up to file the big square
   fill in the upper-left square and the lower-right square
   if the squares are still big enough to draw in, then
      fractal(upper-right square)
      fractal(lower-left square)
   }
```

The image below shows what this might look like after three levels of recursion:



Write a java program to draw the image through eight levels of recursion.

7. Write a recursive method mult (a,b) to multiply two integers *a* and *b* using repeated addition. Test your method with various combinations of numbers. What is the largest number *n* that can be multiplied by 2 using both *mult(2,n)* and mult *(n,2)*?

8. Write a recursive method to determine if a String is a palindrome. Create a String array with several test cases and test your method.

9. Write a program with two methods for finding the nth Fibonacci number, one a recursive method as described in the chapter, and another an iterative method of your design. Create an array with several test cases and test your methods.

10. The French mathematician *Edouard Lucas* made contributions to number theory and to the study of Fibonacci numbers. Lucas invented and marketed the *Tower of Hanoi* puzzle. The *Tower of Hanoi* puzzle appeared in 1883 under the name of *M. Claus* (an anagram of his name Lucas that he often used.) Write an article describing the *Tower of Hanoi* puzzle, the legend Lucas created to go with the puzzle, and a recursive algorithm for solving the puzzle. Include a brief biographical sketch of Lucas.