

Recursion

No discussion of functions or methods is complete without visiting an age-old technique in computer science: *recursion*. A recursive method is a method that calls itself. So, within the definition of the method, there appears a call to the same method. As it turns out, certain computational problems are well-suited to recursion. Any algorithm that computes a result iteratively, where each step builds on a result from the previous step, is a candidate for recursion. We'll illustrate this by developing two example programs.

Factorial of an Integer

A common mathematical operation is to compute the factorial of an integer. If n is an integer, then $n!$ (read " n factorial") is $n \times (n - 1) \times (n - 2) \dots$ computed iteratively until the factor 1 appears. So,

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

By definition, $0! = 1$. Figure 1 shows the listing for Java program that computes the factorial of an integer. The program includes a non-recursive method named `factorial()`.

```
1  import java.io.*;
2
3  public class Factorial
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter a non-negative integer: ");
11         int n = Integer.parseInt(stdin.readLine());
12         if (n < 0)
13         {
14             System.out.println("Oops!");
15             return;
16         }
17         else
18             System.out.println(n + "! = " + factorial(n));
19     }
20
21     public static int factorial(int n)
22     {
23         if (n == 0)
24             return 1;
25         int f = n;
26         for (int i = n - 1; i > 0; i--)
27             f *= i;
28         return f;
29     }
30 }
```

Figure 1. Factorial (non recursive solution)

A sample dialogue with this program follows:

```
PROMPT>java Factorial
Enter a non-negative integer: 9
9! = 362880
```

The method `factorial()` is defined in lines 21-29 and is called in line 18. In the method's signature (line 21), we see that it receives an integer argument, `n`, and returns an integer result. Within the method, the first task is to return immediately with "1" if the argument equals "0" (lines 23-24). Otherwise, a local variable `f` is declared and initialized to `n`. Then, a `for` loop is started with a loop control variable, `i`, initialized to `n - 1`. On each pass through the loop, `i` is decremented. Within the loop `f` is reassigned with its existing value multiplied by `i`. When the loop finishes, `f` holds the factorial of the original integer passed to the method. This value is returned in line 28.

Now let's look at a program with the identical behaviour. `Factorial2` is the same as `Factorial` except the method `factorial()` uses recursion to calculate the factorial of an integer (see Figure 2).

```
1  import java.io.*;
2
3  public class Factorial2
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter a non-negative integer: ");
11         int n = Integer.parseInt(stdin.readLine());
12         if (n < 0)
13             {
14                 System.out.println("Oops!");
15                 return;
16             }
17         else
18             System.out.println(n + "! = " + factorial(n));
19     }
20
21     public static int factorial(int n)
22     {
23         if (n == 0)
24             return 1;
25         else
26             return n * factorial(n - 1);
27     }
28 }
```

Figure 2. `Factorial2.java` (recursive solution)

The definition of `factorial()` in lines 21-27 appears much simpler than in the previous example. It's also a lot trickier to understand because of the recursive approach. Note in line 26 that the method calls itself within its own definition. However, each call passes "one less than" the value of the argument passed in the preceding call; so, with each recursive call, we get closer to the correct answer. A written blow-by-blow explanation is not likely to explain as well as a decent visualization, so let's move directly to Figure 3. The figure illustrates the recursive calls to `factorial()` with an original argument of 4.

Step #	Call #	Call value	Return value
1	1	4	
2	2	3	
3	3	2	
4	4	1	
5	5	0	1
6	4		1×1
7	3		$1 \times 1 \times 2$
8	2		$1 \times 1 \times 2 \times 3$
9	1		$1 \times 1 \times 2 \times 3 \times 4$

Figure 3. Visualization of recursive method calls

The left-hand column shows the steps in sequential time order. The second column identifies each call to `factorial()` by number. The method is called a total of five times. The indentation in the second column suggests the level of recursion. The third column identifies the value of the argument passed to the method and the fourth column identifies the value returned by the method. For each call to `factorial()` the value returned should be the factorial of the value passed. And, indeed, this is the case. For each call number in column two, note that the return value (column four) is the factorial of the call value (column three). The specific case for call #1 is highlighted. The call value was 4, and the return value was $1 \times 1 \times 2 \times 3 \times 4 = 24$.

Reversing Characters in a String - Recursion

Let's explore recursion further with an example of a problem met earlier. The `StringBackwards2` program used a method to reverse the pattern of characters in a string. The `StringBackwards3` program performs the same task except using a recursive method (see Figure 4).

```

1  import java.io.*;
2
3  public class StringBackwards3
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter a message string: ");
11         String s = stdin.readLine();
12         s = backwards(s);
13         System.out.print(s);
14     }
15
16     public static String backwards(String s)
17     {
18         if (s.length() == 1)
19             return s;
20         else
21             {
22                 s = backwards(s.substring(1, s.length())) + s.substring(0, 1);
23                 return s;
24             }
25     }
26 }

```

Figure 4. StringBackwards3.java

The method `backwards()` is defined in lines 16-25. The method receives a `String` argument and returns a reference to a new `String` object. Within its definition, the method calls itself (line 22). In the recursive call, the argument passed is a substring of the original string. The expression

```
s.substring(1, s.length())
```

represents the original string with the first character removed (see line 22). The first character appears in the expression

```
s.substring(0, 1)
```

which is concatenated to the end of the string returned by the `backward()` method. Thus, the original string is gradually reassembled as the recursion unwinds, with the original characters in reverse order. A visualization of this process is presented in the next section.

The Joy of Recursion

Recursion is a topic you either love or hate. For many students, it's a punishing task to develop a recursive solution to a problem that seems pretty simple without recursion. Like a flowcharting assignment, you may find yourself working backwards — by first solving the problem the way you see it, and then retrofitting your solution to meet the requirements of an assignment. And to this, we have no specific remedy to suggest. However, you'll know you are ready to work with recursion — on your terms — when one day while working on a simple iterative problem, you'll think without warning, "Hey, I can do that using recursion." Our advice: Go for it. You'll have a lot of fun, and when you get the recursive solution working, you'll be sold on it.

There are performance tradeoffs to consider when comparing recursive and non-recursive solutions to programming problems. Non-recursive solutions generally use less memory, since answers are "built up" using one set of variables. Recursive solutions must allocate and maintain a set of local variables for each recursive call. The memory space cannot be freed-up until each method returns, and so, more and more memory is allocated as the recursion deepens. There is an additional overhead of calling a method numerous times. Nevertheless, recursion is a topic that persists. For some interesting recursive problems see Chapters 18 and 19 in Hofstadter's *Metamagical Themas* [1].