An Introduction to Recursion, Part 1

By **jmzero**
TopCoder Member

Recursion is a wonderful programming tool. It provides a simple, powerful way of approaching a variety of problems. It is often hard, however, to see how a problem can be approached recursively; it can be hard to "think" recursively. It is also easy to write a recursive program that either takes too long to run or doesn't properly terminate at all. In this article we'll go over the basics of recursion and hopefully help you develop, or refine, a very important programming skill.

What is Recursion?

In order to say exactly what recursion is, we first have to answer "What is recursion?" Basically, a function is said to be recursive if it calls itself. Below is pseudocode for a recursive function that prints the phrase "Hello World" a total of *count* times:

```
function HelloWorld(count)
{
    if(count<1)return
    print("Hello World!")
    HelloWorld(count - 1)
}
```

It might not be immediately clear what we're doing here - so let's follow through what happens if we call our function with *count* set to 10. Since *count* is not less than 1, we do nothing on the first line. On the next, we print "Hello World!" once. At this point we need to print our phrase 9 more times. Since we now have a HelloWorld function that can do just that, we simply call HelloWorld (this time with *count* set to 9) to print the remaining copies. That copy of HelloWorld will print the phrase once, and then call another copy of HelloWorld to print the remaining 8. This will continue until finally we call HelloWorld with *count* set to zero. HelloWorld(0) does nothing; it just returns. Once HelloWorld(0) has finished, HelloWorld(1) is done too, and it returns. This continues all the way back to our original call of HelloWorld(10), which finishes executing having printed out a total of 10 "Hello World!"s.

You may be thinking this is not terribly exciting, but this function demonstrates some key considerations in designing a recursive algorithm:

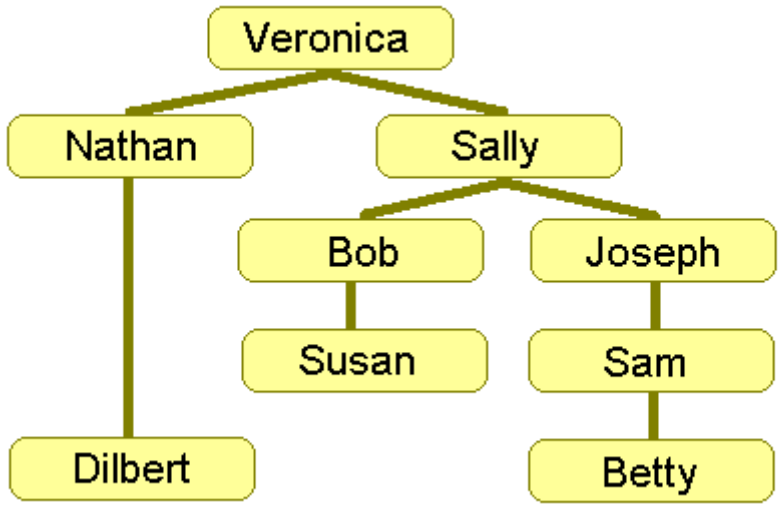
1. **It handles a simple "base case" without using recursion.**
   In this example, the base case is "HelloWorld(0)"; if the function is asked to print zero times then it returns without spawning any more "HelloWorld"s.
2. **It avoids cycles.**
   Imagine if "HelloWorld(10)" called "HelloWorld(10)" which called "HelloWorld(10)." You'd end up with an infinite cycle of calls, and this usually would result in a "stack overflow" error while running. In many recursive programs, you can avoid cycles by having each function call be for a problem that is somehow smaller or simpler than the original problem. In this case, for example, *count* will be smaller and smaller with each call. As the problem gets simpler and simpler (in this case, we'll consider it "simpler" to print something zero times rather than printing it 5 times) eventually it will arrive at the "base case" and stop recursing. There are many ways to avoid infinite cycles, but making sure that we're dealing with progressively smaller or simpler problems is a good rule of thumb.
3. **Each call of the function represents a complete handling of the given task.**
   Sometimes recursion can seem kind of magical in the way it breaks down big problems. However, there is no such thing as a free lunch. When our function is given an argument of 10, we print "Hello World!" once and then we print it 9 more times. We can pass a part of the job along to a recursive call, but the original function still has to account for all 10 copies somehow.

Why use Recursion?

The problem we illustrated above is simple, and the solution we wrote works, but we probably would have been better off just using a loop instead of bothering with recursion. Where recursion tends to shine is in situations where the problem is a little more complex. Recursion can be applied to pretty much any problem, but there are certain scenarios for which you'll find it's particularly helpful. In the remainder of this article we'll discuss a few of these scenarios and, along the way, we'll discuss a few more core ideas to keep in mind when using recursion.

Scenario #1: Hierarchies, Networks, or Graphs

In algorithm discussion, when we talk about a graph we're generally not talking about a chart showing the relationship between variables (like your TopCoder ratings graph, which shows the relationship between time and your rating). Rather, we're usually talking about a network of things, people, or concepts that are connected to each other in various ways. For example, a road map could be thought of as a graph that shows cities and how they're connected by roads. Graphs can be large, complex, and awkward to deal with programatically. They're also very common in algorithm theory and algorithm competitions. Luckily, working with graphs can be made much simpler using recursion. One common type of a graph is a hierarchy, an example of which is a business's organization chart:



Name	Manager
Betty	Sam
Bob	Sally
Dilbert	Nathan
Joseph	Sally
Nathan	Veronica
Sally	Veronica
Sam	Joseph
Susan	Bob
Veronica	

In this graph, the objects are people, and the connections in the graph show who reports to whom in the company. An upward line on our graph says that the person lower on the graph reports to the person above them. To the right we see how this structure could be represented in a database. For each employee we record their name and the name of their manager (and from this information we could rebuild the whole hierarchy if required - do you see how?).

Now suppose we are given the task of writing a function that looks like "countEmployeesUnder(employeeName)". This function is intended to tell us how many employees report (directly or indirectly) to the person named by *employeeName*. For example, suppose we're calling "countEmployeesUnder('Sally')" to find out how many employees report to Sally.

To start off, it's simple enough to count how many people work directly under her. To do this, we loop through each database record, and for each employee whose manager is Sally we increment a counter variable. Implementing this approach, our function would return a count of 2: Bob and Joseph. This is a start, but we also want to count people like Susan or Betty who are lower in the hierarchy but report to Sally indirectly. This is awkward because when looking at the individual record for Susan, for example, it's not immediately clear how Sally is involved.

A good solution, as you might have guessed, is to use recursion. For example, when we encounter Bob's record in the database we don't just increment the counter by one. Instead, we increment by one (to count Bob) and then increment it by the number of people who report to Bob. How do we find out how many people report to Bob? We use a recursive call to the function we're writing: "countEmployeesUnder('Bob')". Here's pseudocode for this approach:

```
function countEmployeesUnder(employeeName)
{
    declare variable counter
    counter = 0
    for each person in employeeDatabase
    {
        if(person.manager == employeeName)
        {
            counter = counter + 1
            counter = counter + countEmployeesUnder(person.name)
        }
    }
    return counter
}
```

If that's not terribly clear, your best bet is to try following it through line-by-line a few times mentally. Remember that each time you make a recursive call, you get a new copy of all your local variables. This means that there will be a separate copy of *counter* for each call. If that wasn't the case, we'd really mess things up when we set counter to zero at the beginning of the function. As an exercise, consider how we could change the function to increment a global variable instead. Hint: if we were incrementing a global variable, our function wouldn't need to return a value.

Mission Statements

A very important thing to consider when writing a recursive algorithm is to have a clear idea of our function's "mission statement." For example, in this case I've assumed that a person shouldn't be counted as reporting to him or herself. This means "countEmployeesUnder('Betty')" will return zero. Our function's mission statment might thus be "Return the count of people who report, directly or indirectly, to the person named in *employeeName* - not including the person named *employeeName*."

Let's think through what would have to change in order to make it so a person did count as reporting to him or herself. First off, we'd need to make it so that if there are no people who report to someone we return one instead of zero. This is simple -- we just change the line "counter = 0" to "counter = 1" at the beginning of the function. This makes sense, as our function has to return a value 1 higher than it did before. A call to "countEmployeesUnder('Betty')" will now return 1.

However, we have to be very careful here. We've changed our function's mission statement, and when working with recursion that means taking a close look at how we're using the call recursively. For example, "countEmployeesUnder('Sam')" would now give an incorrect answer of 3. To see why, follow through the code: First, we'll count Sam as 1 by setting counter to 1. Then when we encounter Betty we'll count her as 1. Then we'll count the employees who report to Betty -- and that will return 1 now as well.

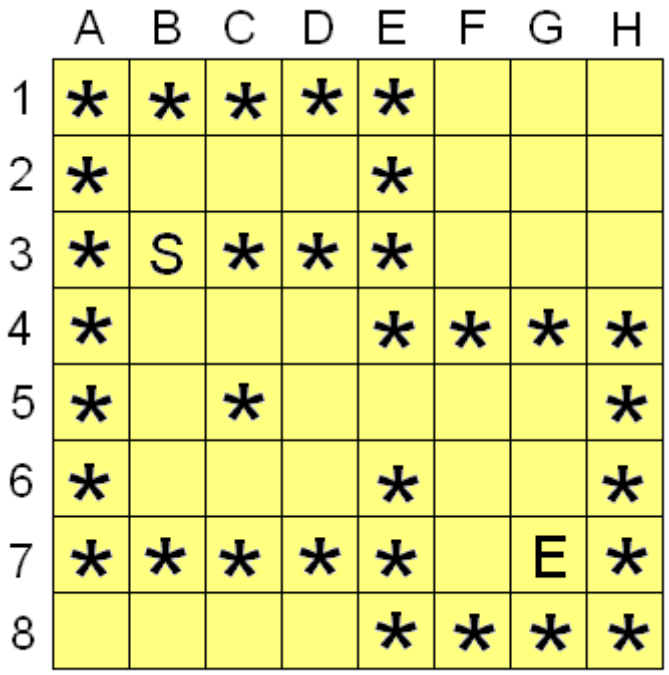
It's clear we're double counting Betty; our function's "mission statement" no longer matches how we're using it. We need to get rid of the line "counter = counter + 1", recognizing that the recursive call will now count Betty as "someone who reports to Betty" (and thus we don't need to count her before the recursive call).

As our functions get more and more complex, problems with ambiguous "mission statements" become more and more apparent. In order to make recursion work, we must have a very clear specification of what each function call is doing or else we can end up with some very difficult to debug errors. Even if time is tight it's often worth starting out by writing a comment detailing exactly what the function is supposed to do. Having a clear "mission statement" means that we can be confident our recursive calls will behave as we expect and the whole picture will come together correctly.

Scenario #2: Multiple Related Decisions

When our program only has to make one decision, our approach can be fairly simple. We loop through each of the options for our decision, evaluate each one, and pick the best. If we have two decisions, we can have nest one loop inside the other so that we try each possible combination of decisions. However, if we have a lot of decisions to make (possibly we don't even know how many decisions we'll need to make), this approach doesn't hold up.

For example, one very common use of recursion is to solve mazes. In a good maze we have multiple options for which way to go. Each of those options may lead to new choices, which in turn may lead to new choices as the path continues to branch. In the process of getting from start to finish, we may have to make a number of related decisions on which way to turn. Instead of making all of these decisions at once, we can instead make just one decision. For each option we try for the first decision, we then make a recursive call to try each possibility for all of the remaining decisions. Suppose we have a maze like this:



For this maze, we want to determine the following: is it possible to get from the 'S' to the 'E' without passing through any '*' characters. The function call we'll be handling is something like this: "isMazeSolveable(maze[ ][ ])". Our maze is represented as a 2 dimensional array of characters, looking something like the grid above. Now naturally we're looking for a recursive solution, and indeed we see our basic "multiple related decision" pattern here. To solve our maze we'll try each possible initial decision (in this case we start at B3, and can go to B2 or B4), and then use recursion to continue exploring each of those initial paths. As we keep recursing we'll explore further and further from the start. If the maze is solveable, at some point we'll reach the 'E' at G7. That's one of our base cases: if we are asked "can we get from G7 to the end", we'll see that we're already at the end and return

true without further recursion. Alternatively, if we can't get to the end from either B2 or B4, we'll know that we can't get to the end from B3 (our initial starting point) and thus we'll return false.

Our first challenge here is the nature of the input we're dealing with. When we make our recursive call, we're going to want an easy way to specify where to start exploring from - but the only parameter we've been passed is the maze itself. We could try moving the 'S' character around in the maze in order to tell each recursive call where to start. That would work, but would be very slow because in each call we'd have to first look through the entire maze to find where the 'S' is. A better idea would be to find the 'S' once, and then pass around our starting point in separate variables. This happens fairly often when using recursion: we have to use a "starter" function that will initialize any data and get the parameters in a form that will be easy to work with. Once things are ready, the "starter" function calls the recursive function that will do the rest of the work. Our starter function here might look something like this:

```
function isMazeSolveable(maze[][])

{

    declare variables x,y,startX,startY

    startX=-1

    startY=-1



    // Look through grid to find our starting point

    for each x from A to H

    {

        for each y from 1 to 8

        {

            if maze[x][y]=='S' then

            {

                startX=x

                startY=y

            }

        }

    }



    // If we didn't find starting point, maze isn't solveable

    if startX==-1 then return false



    // If we did find starting point, start exploring from that point

    return exploreMaze(maze[][],startX,startY)

}
```

We're now free to write our recursive function exploreMaze. Our mission statement for the function will be "Starting at the position (X,Y), is it possible to reach the 'E' character in the given maze. If the position (x,y) is not traversable, then return false." Here's a first stab at the code:

```
function exploreMaze(maze[][],x,y)
{
```

```
    // If the current position is off the grid, then
    // we can't keep going on this path
    if y>8 or y<1 or x<'A' or x>'H' then return false

    // If the current position is a '*', then we
    // can't continue down this path
    if maze[x][y]=='*' then return false

    // If the current position is an 'E', then
    // we're at the end, so the maze is solveable.
    if maze[x][y]=='E' then return true

    // Otherwise, keep exploring by trying each possible
    // next decision from this point.  If any of the options
    // allow us to solve the maze, then return true.  We don't
    // have to worry about going off the grid or through a wall -
    // we can trust our recursive call to handle those possibilities
    // correctly.
    if exploreMaze(maze,x,y-1) then return true  // search up
    if exploreMaze(maze,x,y+1) then return true  // search down
    if exploreMaze(maze,x-1,y) then return true  // search left
    if exploreMaze(maze,x+1,y) then return true  // search right

    // None of the options worked, so we can't solve the maze
    // using this path.
    return false
}
```

Avoiding Cycles

If you're keen eyed, you likely noticed a flaw in our code above. Consider what happens when we're exploring from our initial position of B3. From B3, we'll try going up first, leading us to explore B2. From there, we'll try up again and go to B1. B1 won't work (there's a '*' there), so that will return false and we'll be back considering B2. Since up didn't work, we'll try down, and thus we'll consider B3. And from B3, we'll consider B2 again. This will continue on until we error out: there's an infinite cycle.

We've forgotten one of our rules of thumb: we need to make sure the problem we're considering is somehow getting smaller or simpler with each recursive call. In this case, testing whether we can reach the end from B2 is no simpler than considering whether we can reach the end from B3. Here we can get a clue from real-life mazes: if you feel like you've seen this place before, then you may be going in circles. We need to revise our mission statement to include "avoid exploring from any position we've already considered". As the number of places we've considered grows, the problem gets simpler and simpler because each decision will have less valid options.

The remaining problem is, then, "how do we keep track of places we've already considered?". A good solution would be to pass around another 2 dimensional array of true/false values that would contain a "true" for each grid cell we've already been to. A quicker-and-dirtier way would be to change *maze* itself, replacing the current position with a '*' just before we make any recursive calls. This way, when any future path comes back to the point we're considering, it'll know that it went in a circle and doesn't need to continue exploring. Either way, we need to make sure we mark the current point as visited before we make the recursive calls, as otherwise we won't avoid the infinite cycle.

Scenario #3: Explicit Recursive Relationships

You may have heard of the Fibonacci number sequence. This sequence looks like this: 0, 1, 1, 2, 3, 5, 8, 13... After the first two values, each successive number is the sum of the previous two numbers. We can define the Fibonacci sequence like this:

```
    Fibonacci[0] = 0
    Fibonacci[1] = 1
    Fibonacci[n] = Fibonacci[n-2] + Fibonacci[n-1]
```

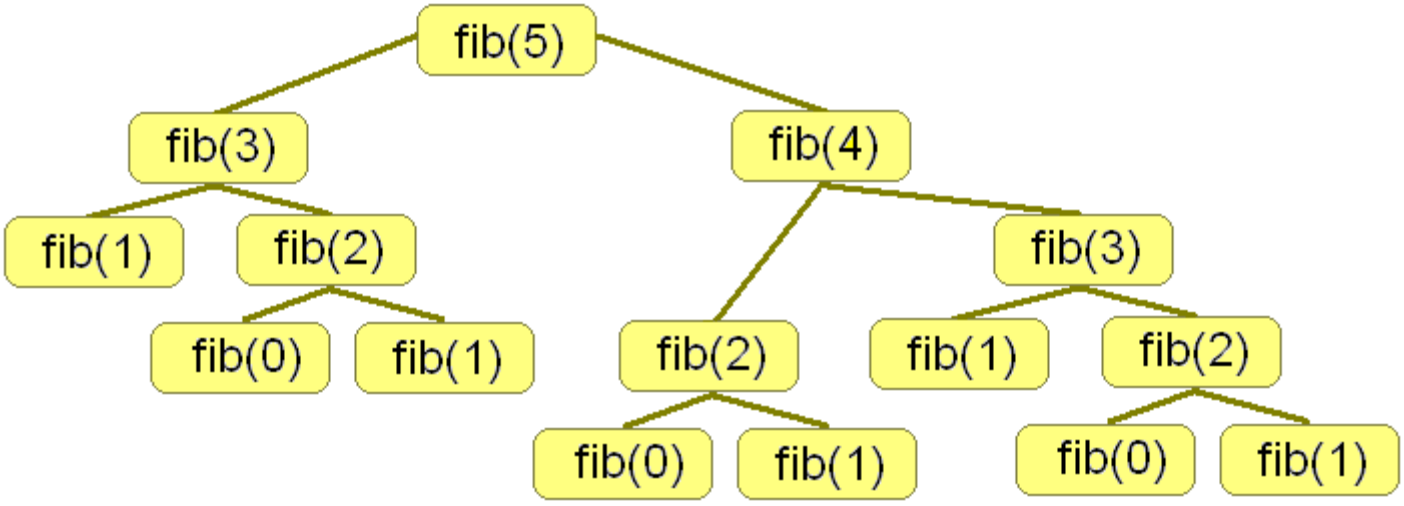
This definition already looks a lot like a recursive function. 0 and 1 are clearly the base cases, and the other possible values can be handled with recursion. Our function might look like this:

```
function fib(n)
{
    if(n<1)return 0
    if(n==1)return 1
    return fib(n-2) + fib(n-1)
}
```

This kind of relationship is very common in mathematics and computer science - and using recursion in your software is a very natural way to model this kind of relationship or sequence. Looking at the above function, our base cases (0 and 1) are clear, and it's also clear that *n* gets smaller with each call (and thus we shouldn't have problems with infinite cycles this time).

Using a Memo to Avoid Repetitious Calculation

The above function returns correct answers, but in practice it is extremely slow. To see why, look at what happens if we called "fib(5)". To calculate "fib(5)", we'll need to calculate "fib(4)" and "fib(3)". Each of these two calls will make two recursive calls each - and they in turn will spawn more calls. The whole execution tree will look like this:



The above tree grows exponentially for higher values of *n* because of the way calls tend to split - and because of the tendency we have to keep re-calculating the same values. In calculating "fib(5)", we ended up calculating "fib(2)" 3 times. Naturally, it would be better to only calculate that value once - and then remember that value so that it doesn't need to be calculated again next time it is asked for. This is the basic idea of memoization. When we calculate an answer, we'll store it in an array (named *memo* for this example) so we can reuse that answer later. When the function is called, we'll first check to see if we've already got the answer stored in *memo*, and if we do we'll return that value immediately instead of recalculating it.

To start off, we'll initialize all the values in *memo* to -1 to mark that they have not been calculated yet. It's convenient to do this by making a "starter" function and a recursive function like we did before:

```
function fib(n)
{
    declare variable i,memo[n]

    for each i from 0 to n
    {
        memo[i]=-1
    }
    memo[0]=0
    memo[1]=1

    return calcFibonacci(n,memo)
}

function calcFibonacci(n,memo)
{
    // If we've got the answer in our memo, no need to recalculate
    if memo[n]!=-1 then return memo[n]

    // Otherwise, calculate the answer and store it in memo
    memo[n] = calcFibonacci(n-2,memo) + calcFibonacci(n-1,memo)

    // We still need to return the answer we calculated
    return memo[n]
}
```

The execution tree is now much smaller because values that have been calculated already no longer spawn more recursive calls. The result is that our program will run much faster for larger values of *n*. If our program is going to calculate a lot of Fibonacci numbers, it might be best to keep *memo* somewhere more persistent; that would save us even more calculations on future calls. Also, you might have noticed another small trick in

the above code. Instead of worrying about the base cases inside *calcFibonacci*, we pre-loaded values for those cases into the memo. Pre-loading base values - especially if there's a lot of them - can make our recursive functions faster by allowing us to check base cases and the memo at the same time. The difference is especially noticeable in situations where the base cases are more numerous or hard to distinguish.

This basic memoization pattern can be one of our best friends in solving TopCoder algorithm problems. Often, using a memo is as simple as looking at the input parameters, creating a *memo* array that corresponds to those input parameters, storing calculated values at the end of the function, and checking the memo as the function starts. Sometimes the input parameters won't be simple integers that map easily to a *memo* array - but by using other objects (like a hash table) for the *memo* we can continue with the same general pattern. In general, if you find a recursive solution for a problem, but find that the solution runs too slowly, then the solution is often memoization.

Conclusion

Recursion is a fundamental programming tool that can serve you well both in TopCoder competitions and "real world" programming. It's a subject that many experienced programmers still find threatening, but practice using recursion in TopCoder situations will give you a great start in thinking recursively, and using recursion to solve complicated programming problems.