Recursion in C++

While recursion doesn't necessarily introduce any new programming language syntax, it's a difficult topic and often given multiple weeks of attention in collegiate computer science courses. There's no way anyone will master recursion with this one lecture, but the goal of this lecture is to give students enough background information for the brute force lecture that follows, since brute force (combinations, permutations) is a common technique used in programming competitions.

By definition, a recursive function is a function that sometimes calls itself. The reason for "sometimes" is if a recursive function always called itself, there would be no end to the chain of function calls. Perhaps the easiest way to see recursion is via a short example.

First Example: Factorial Function

One of the most common examples used in programming courses is the factorial function ("n!" is read as "n factorial"):

n! = 1 x 2 x 3 x … x (n – 1) x n

If we look at this definition carefully, we could write it as follows (so long as n is positive and we define 0! = 1):

n! = (n – 1)! x n

Correspondingly, here is the code for the factorial function in C++. (Note: ll is typdefed as for long long in the example code.)

```
ll fact(int n) {
    if (n == 0) return 1;
    return n*fact(n-1);
}
```

For example, when we call fact(3), it will call fact(2), which calls fact(1), which calls fact(0). At that moment in time, the call stack looks like this:

fact(0)
fact(1)
fact(2)
fact(3)
main

When fact(0) is called, it returns 1 back to fact(1), which called it. This allows fact(1) to complete and return 1 to fact(2). Then fact(2) completes and returns 2 to fact(3). Finally, fact(3) can then complete, returning 3*2 = 6 to main. (In this example we assumed that fact(3) was directly called from main.)

Here is a picture of the process over time as each recursive call finishes:

fact(1) → returns 1 x 1 (from fact(0))
fact(2) → returns 2 x 1 (from fact(1))
fact(3) → returns 3 x 2 (from fact(2))
There are two keys to this code working:

1. Breaking down the original problem into subproblems such that one (or more) of the subproblems is a problem of the exact same nature, but smaller.

2. Figuring out a case that is so small there is no longer any need to break it into subproblems. It can be solved very easily on its own. This is called the base case.

In this case, the factorial function was fairly straightforward to break down, but part of what makes recursion difficult is figuring out how to state a problem so that it can be solved with subproblems of the exact same nature.

Base cases are usually fairly easy to figure out. Think about small enough cases and try to just directly solve them. Typically, base cases come first in the layout of a recursive function because these are the cases when no recursive call (call to the same function) are made, so we typically want to get these out of the way before we try to call the function recursively.

<u>Other Mathematical Examples: Triangle Numbers, Power</u>
The nth Triangle Number is the sum of the first n positive integers. Accordingly, the recursive function looks very much like the factorial function:

```
ll trinum(int n) {
    if (n == 0) return 0;
    return n + trinum(n-1);
}
```

Here we see that the additive identity is 0 (not 1) and of course, in the return step, we are performing addition instead of multiplication.

We can also think of exponentiation recursively. Utilizing a similar breakdown to the factorial function, we find that

$b^e = b \times b^{e-1}$, so long as $e > 0$. If e is 0, then the result is 1 (multiplicative identity).

Utilizing this mathematical breakdown, we translate to get the following code:

```
ll mypow(ll base, int exp) {
    if (exp == 0) return 1;
    return base*mypow(base, exp-1);
}
```

One thing to notice here is that each of these functions isn't necessarily all that efficient. Each can easily be written with a for loop and with a little math, we can rewrite the second one without recursion or loops; just a single return statement that is a simple function in terms of n. However, the purpose of these examples is to teach the technique of recursion. Now, we'll use the last example to build up a more efficient example which shows the power of recursion.

Fast Modular Exponentiation

A common task in competitive programming is modular exponentiation. Given a base, b, an exponent, e, and a modulus value, m, compute the remainder when b^e is divided by m. One key rule with making this calculation is that you can do intermediate modulus steps as much as you want without affecting the answer. So for example, b^3 % m is the same as ((b^2 % m)*b)%m. The latter seems like more work, but in reality it's less work!!! The reason is the sheer size of a number with a large exponent may be many digits long, but we are guaranteed that any number mod m is less than m. Thus, if we always do an intermediate modulus operation after every multiply, we'll never have an intermediate number greater than m^2. Thus, even if b^e would be astronomical, we can calculate b^e % m without ever storing a number greater than m^2. Once we understand this idea, we can come up with much more efficient code to calculate b^e % m than the previous example.

We need a new recursive breakdown to the function. Our old one:

$b^e = b \times b^{e-1}$

is slow because it requires at total of e recursive calls in sequence, similar to a for loop that runs e times.

But we can utilize one key idea here. If e is even, then we can rewrite our formula differently:

$$b^e = (b^{e/2}) \times (b^{e/2})$$

Now, add the mod in there and we have:

$$b^e \% m = ((b^{e/2} \% m) \times (b^{e/2} \% m)) \% m$$

If the exponent is odd, we can use our previous breakdown. But, if you think about, it, in one recursive call, if the exponent is odd, then in the next one, it will be even.

Here's the code (we assume mod > 1):

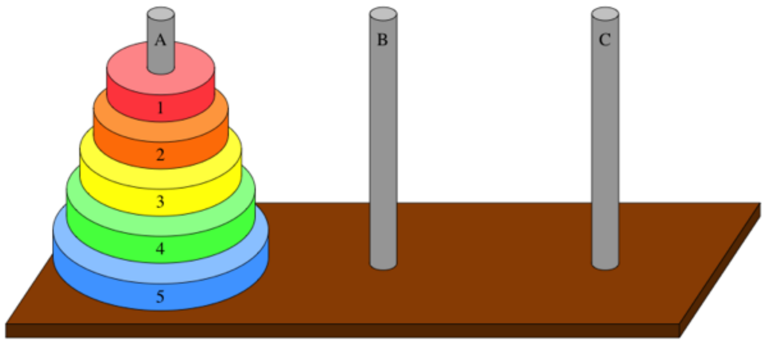
```
ll fastmodpow(ll base, ll exp, ll mod) {

    if (exp == 0) return 1;

    if (exp%2 == 0) {
        ll tmp = fastmodpow(base, exp/2, mod);
        return (tmp*tmp)%mod;
    }

    return (base*fastmodpow(base, exp-1, mod))%mod;
}
```

The beauty of this code is that once we know the answer to $b^{e/2}$ % m, we don't have to recalculate it a second time (there's only one recursive call in the second if). We reuse that answer, multiply it by itself and mod, in some sense, saving a long loop to recalculate that same value.

Even with an exponent as large as 10^9, this code is guaranteed not to make more than about 60 recursive calls. This is astonishingly better than the 10^9 operations the old recursive code (or for loop) would do.

Towers of Hanoi

A famous puzzle involves three towers with a series of disks (largest on the bottom, smallest on the top), stacked on a single tower. The goal is to move all of the disks from the original tower to one of the other towers. The restrictions are that only one disk can be moved at a time, and at no point can a larger disk be placed on a smaller disk.



For this picture, imagine that we label the towers 1, 2 and 3 from left to right. (The disks are labeled from 1 to n, where 1 is the smallest and n is the largest.) Let's say we want to move all the n disks from tower startT to tower endT in general and that the function to solve this task (let's just say the function prints out all the moves to solve the puzzle), has the following prototype:

```
void towers(int n, int startT, int endT);
```

First of all, we know that at some point, we must move the bottom disk, disk n. A good question to ask is, when will be able to do so. (We definitely can't at the beginning of the puzzle!)

In order for disk n to move, disks 1 through n – 1 must be moved to a different tower and disk 5 must be alone on startT. (This is because it's bigger than all of the other disks and it can't ever be anything but the bottom disk of any tower, and we can only ever move the top disk on any tower.)

BUT…if disks 1 through n – 1 occupy BOTH of the other towers, then disk n can NOT move because disk n MUST move from one empty tower to another empty tower.

It stands to reason that when we move disk n, we must have its tower with nothing else on it, AND the tower it's moving to have nothing on it. The ONLY way to create this situation is to move ALL the disks 1 – (n-1) from tower startT to the "other" tower (the one that is neither startT nor endT). But, if you think about it, this is really a smaller problem of the exact same nature – RECURSION! So, now we have a algorithmic sketch of our partial solution:

```
towers(int n, int startT, int endT)
Step 1. towers(n-1, startT, "other tower")
Step 2. Move disk n from tower startT to tower endT.
```

Thus, our picture now looks something like this:

```
                                         1
                                         2
                                        …
                         n              n-1
___________          __________       __________
Start Tower          End Tower        Other Tower
```

At this point it should be clear that all we have left to do is recursively move the first n – 1 disks from "Other Tower" to "End Tower", so that our complete algorithm is:

```
towers(int n, int startT, int endT)
Step 1. towers(n-1, startT, "other tower")
Step 2. Move disk n from tower startT to tower endT.
Step 3. towers(n-1, "other tower", endT)
```

An awful lot of what's above looks like code except for step 2 and a reference to "other tower." To calculate "other tower", note that the sum of the three tower numbers is 1 + 2 + 3 = 6 and that we can calculate the number of the other tower like so:

```
int otherT = 6 - startT - endT;
```

Finally, for step 2, we'll just do a print statement (cout). Here's the function:

```
void towers(int n, int startT, int endT) {

     // No work to be done.
     if (n == 0) return;

     // Calculate the number of the third tower.
     int otherT = 6 - startT - endT;

     // Move the first n-1 disks out of the way.
     towers(n-1, startT, otherT);

     // Move the bottom disk to its end spot.
     cout << "Move disk " << n << " from tower " << startT << "
to tower " << endT << endl;

     // Move the first n-1 disks to their final spot.
     towers(n-1, otherT, endT);

}
```

The only addition to this code is the base case (n == 0), where no work has to be done, so we simply return before we do any recursion here.

Binary Search

Note that C++ has a built in binary search which will return true or false indicating if an element exists in a sorted vector or not. But, binary search is such an important technique and so adaptable and applicable that it's best to learn how to do it from scratch. This algorithm can be written iteratively equally easily as recursively.

The problem to be solved is as follows: given a sorted vector and a value to search for, determine if that value is in the array or not. Here is an example:

Vector: 2,   3,   6,   22,   39,   41,   52

Search Value: 41

One way to find the value is to do a for loop through the data. But, it seems like a waste of time because we are not using the fact the array is sorted. (If the array were much better, it would be an even bigger time waste!) The key idea is that if we look halfway through the array (in this case index 3), and we compare the value we're looking for (in this case comparing 41 to 22), then we can immediately find out if we should look to the right or the left. (There will never be a reason to look to both sides!) It makes sense for us to minimize the maximum size of right or left (the maximum range of our next search). We can do this splitting the sides roughly in half.

In short, a search for 41 in the array from index 0 to index 7 has resulted in a new recursive search for 41 in the array from index 4 to index 7, since we now have proof that index 3 is too small.

Thus, when making a comparison to the middle element, one of three things will happen:

1. The value is on the left side, so we must recursively search for it from the left index to the middle index minus 1.

2. The value is on the right side, so we must recursively search for it from the middle index plus 1 to the right index.

3. The value is where we looked, we can return true and do no further work!

The final consideration is that our search space is empty (for example we want to find the item in the subarray starting at index 7 and ending at index 6.) In this case, we just return false.

The final code looks like this:

```
bool binarysearch(const vector<int>& items, int value, int low,
int high) {

    // No search range we return false.
    if (low>high) return false;

    // This is halfway between low and high.
    int mid = (low+high)/2;

    // Looking for something small, go left.
    if (value < items[mid])
        return binarysearch(items, value, low, mid-1);

    // Looking for something big, go right.
    else if (value > items[mid])
        return binarysearch(items, value, mid+1, high);

    // We found it!
    else
        return true;
}
```

Floodfill

The original idea and name “floodfill” comes from the Microsoft Paint paintbucket tool that fills an enclosed area with a color. The basic idea is that we’re on some 2 dimensional space and at one point (or grid square), a “flood” (or leak) emanates from it. It goes in “all directions” and only stops when there’s a boundary set up to block it. Here is an example where the boundary squares are marked as ‘*’ and the location where the flood starts is marked by ‘~’:

```
**************
** ~***  *** *
*    *   *   *
*    *   *****
*        *   *
*    *   *   *
**************
*   *****    *
*    *       *
*   *        *
**************
```

After executing the floodfill, we should have this:

```
**************
**~~***~~*** *
*~~~~*~~~*   *
*~~~~*~~~*****
*~~~~~~~~*   *
*~~~~*~~~*   *
**************
*   *****    *
*    *       *
*   *        *
**************
```

For this example, we assume that the water (‘~’) only flows up, down, left and right. Let’s think about how to solve this. Imagine that our flood starts at coordinate (x, y). Then, in some sense, we want to recursively flood locations of the form:

(x + dx, y + dy)

where (dx, dy) represents a valid direction the water can move “in one step.” The exceptions would be if

(a) The new location is off our grid

(b) The new location is a boundary square which stops the flood

(c) The new location was previously flooded.

If location (x, y) recursively floods location (x + dx, y + dy) and then location (x + dx, y + dy) follows this by recursively flooding location (x, y), then this chain will never stop (infinite recursion). So, one of our base cases (OR conditions when we never call the recursion) has to be based on whether or not we previously flooded that location.

DR/DC or DX/DY arrays

There are MANY competitive programming problems on grids, and most of them describe some sort of valid "movement." To make code for grids more concise and easier to debug, it's critical to use DR/DC arrays which store the definition of movement for the particular problem at hand. (In Counting Stars, which we'll see in a moment, we can move up, down, left and right.) Shockingly, I've seen DR/DC arrays of size 2, 4, 6 and 8, so quite a few varieties and often times they are just problem specific. The good news is that the DR/DC framework allows us to handle all cases in a near identical way while saving quite a few lines of code compared to the alternative. Here's valid DR/DC arrays for up, down, left and right movement:

```
const vector<int> DR = {-1,0,0,1};
const vector<int> DC = {0,-1,1,0};
```

(Note: In most of my posted materials I use DX/DY, but when I coded this example, I used DR and DC, where rows are horizontal and the first index and columns are vertical and the second index.)

The reason this is so handy is that we can easily loop through the neighbors of a location in row x, column y as follows:

```
for (int i=0; i<DR.size(); i++) {
    int newx = x + DR[i];
    int newy = y + DC[i];
    // Process location (newx, newy)
}
```

Let's investigate how this works. Consider the case that x = 2, y = 3. Look at this chart for the values produced for newx and newy in the loop

direction	i	x	y	DR[i]	DC[i]	newx	newy
up	0	2	3	-1	0	2-1=1	3+0=3
left	1	2	3	0	-1	2+0=2	3-1=2
right	2	2	3	0	1	2+0=2	3+1=4
down	3	2	3	1	0	2+1=3	3+0=3

Thus, the four ordered pairs produced by this loop for (newx, newy) are (1, 3), (2, 2), (2, 4), and (3, 3) respectively. These are precisely the grid squares that are above, to the left, to the right and below location (2, 3).

Two Archtypes for floodfill

This leads us to two archetypes for floodfill:

```
void fill(vector<string> grid, int x, int y) {

    grid[x][y] = MARKED;

    for int i=0; i<DR.size(); i++) {
        int newx = x + DR[i];
        int newy = y + DC[i];

        if (!inbounds(newx, newy)) continue;
        if (grid[newx][newy] != FILL) continue;

        fill(grid, newx, newy);
    }

}

void fill(vector<string> grid, int x, int y) {

    if (!inbounds(x,y) return;
    if (grid[x][y] != FILL) return;

    grid[x][y] = MARKED;

    for int i=0; i<DR.size(); i++) {
        int newx = x + DR[i];
        int newy = y + DC[i];
        fill(grid, newx, newy);
    }

}
```

In short, there are two ways to avoid recursing when you shouldn’t: not calling the recursion on squares that you shouldn’t, OR, screening away invalid cases in the base case before you try recursing.

Kattis Problem: Counting Stars

In this problem you are given a grid with either black sky or white stars as a 2D character grid. Here is the problem description:

https://open.kattis.com/problems/countingstars

All of the black sky is indicated by the character '#' and all of the white stars are indicated by the character '-'. The goal of the problem is to identify the number of stars in the grid. A single star is a connected area of '-' characters. So the basic method of solution is to loop through the whole grid. When an unfilled '-' character is reached, we run our floodfill on the region its in and add 1 to our total count. When we run the fill, the '-' characters are changed, so that we don't count them for a different star. Here is the fill function used to solve the problem where FILL = '*' and STAR = '-':

```
// Fill the star at (myr, myc).
void fillStar(int myr, int myc) {

    // Fill this square.
    grid[myr][myc] = FILL;

    // Try all directions.
    for (int i=0; i<DR.size(); i++) {

        // Skip stuff out of bounds and previously filled.
        if (!inbounds(myr+DR[i],myc+DC[i])) continue;
        if (grid[myr+DR[i]][myc+DC[i]] != STAR) continue;

        // Recursively fill this star.
        fillStar(myr+DR[i],myc+DC[i]);
    }
}
```

Once we have this, it’s fairly easy to write the rest of the support code which reads in each grid and processes the cases:

```
int main() {

    // Process cases.
    int cnt = 1;
    while (cin >> r) {

        // Gotta clear the grid...
        cin >> c;
        grid.clear();

        // Read it.
        for (int i=0; i<r; i++) {
            string tmp;
            cin >> tmp;
            grid.push_back(tmp);
        }

        // Solve it!
        cout << "Case " << cnt++ << ": " << solve() << endl;
    }

    return 0;
}

// Returns true iff (myr,myc) is in the grid.
bool inbounds(int myr, int myc) {
    return myr>=0 && myr<r && myc>=0 && myc<c;
}

// Solves the given input case.
int solve() {
    int res = 0;

    // Go to each square.
    for (int i=0; i<r; i++) {
        for (int j=0; j<c; j++) {

            // If necessary, fill this star.
            if (grid[i][j] == STAR) {
                res++;
                fillStar(i, j);
            }
        }
    }

    // Ta da!
    return res;
}
```