Recursion Review

Here we present a short review of recursion and in particular highlight the role of the stack. This is preparation for various maze algorithms we will present later that requires a good understanding of recursion.

A recursive solution consists of solving a smaller version of the same problem. Eventually the problem gets so small that we can just solve it directly. This is call the base case, or termination case, which ends recursion.

Here is the familiar recursive function that returns x factorial:

Definition: $x! = x*(x-1)*(x-2)*(x-3)* \ldots * 1$

$0! = 1$

$1! = 1$

$2! = 2 * 1! = 2 * 1$

$3! = 3 * 2! = 3 * (2*1)$

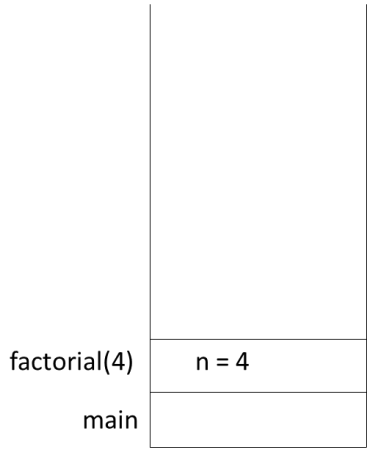
$4! = 4 * 3! = 4 * (3 * (2 * 1))$

$n! = n * (n - 1)!$

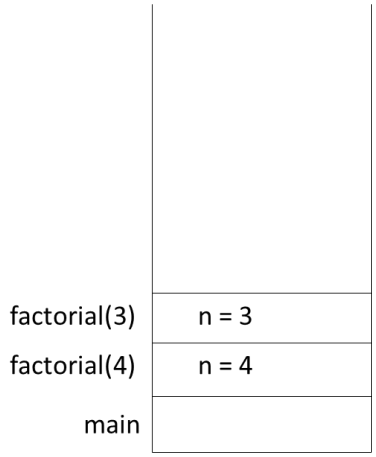
```
long factorial(int n)
{
        if (n == 0) return 1;
        else
        {
                return n * factorial(n-1);
        }
}

int main()
{
  cout << factorial(4) << endl;
}
```

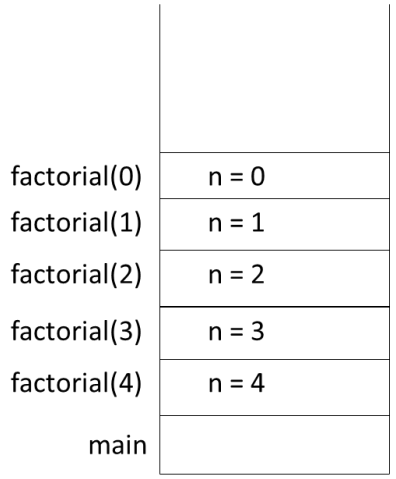
Let's see this program run in the context of the stack. When we start there is a stack frame for main. When main calls factorial it sends in the value 4 for n. We create a new stack frame for the factorial function with n = 4.



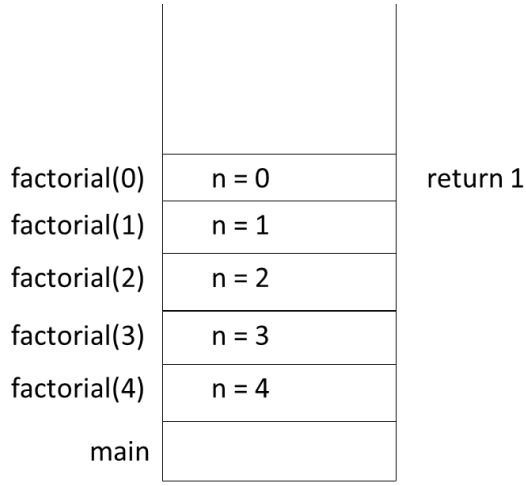
Since n is not zero the factorial(n = 4) function will return 4 * factorial(3). The call to factorial(3) creates a new frame on the stack:



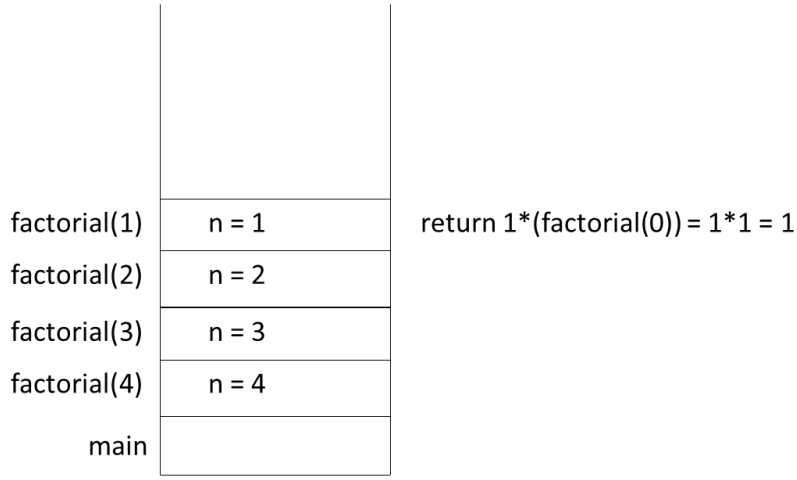
We keep calling factorial and pushing stack frames onto the stack for each factorial call until we send in the value 0:



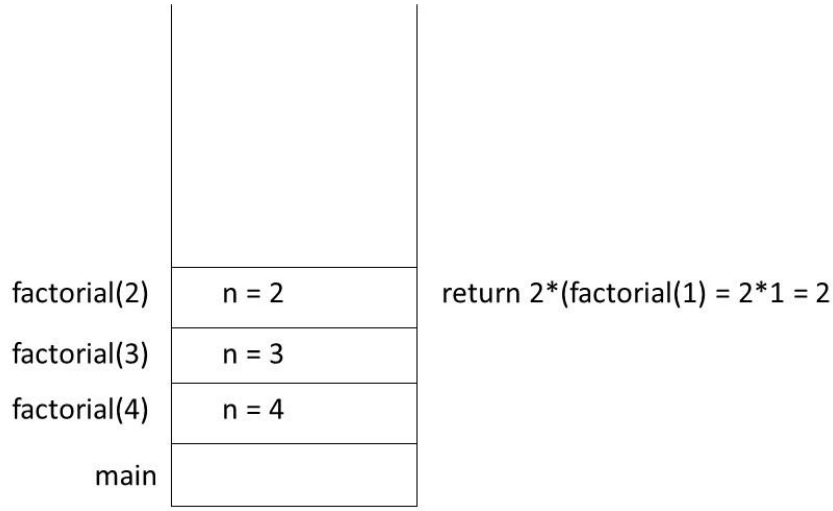
At this point we execute the code “return 1” and remove the stack frame for factorial (n = 0).



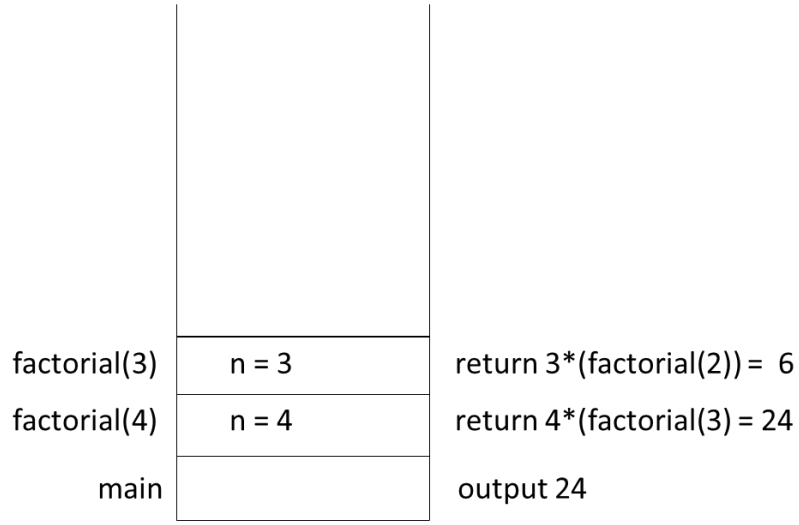
Back in the stack frame for factorial (n = 1) we now return n * factorial(n-1) which in this case is 1 * factorial(0) = 1*1 = 1.



The stack frame for factorial(n = 1) is now removed and computation for returning n*factorial(n-1) is completed in the context of the stack frame for factorial(n = 2).



The process repeats until we get back to main:



More compactly we can depict the sequence of recursive calls as:

factorial(4)

return 4*factorial(3)

return 3*factorial(2)

return 2*factorial(1)

return 1 * factorial(0)

return 1

return 1*1 =1

return 2*1 = 2

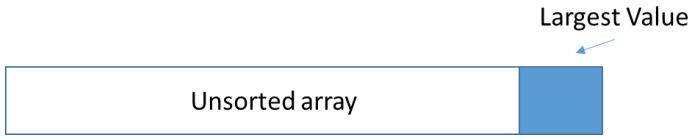
return 3*2 = 6

return 4*6 = 24

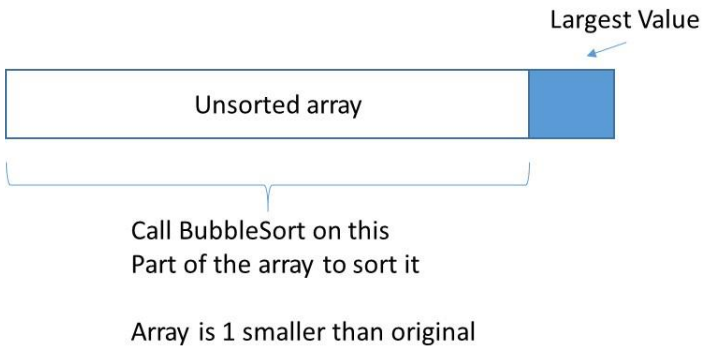
The stack is key to making recursion work because without it we wouldn't get a different copy of the variable n for each recursive call. If all the recursive calls shared the same variable then recursion would not work.

Since we just talked about sorting let's make a recursive version of Bubble Sort. In Bubble Sort we do the following:

1. Bubble the largest value to the end of the array



2. Repeat the process but pretend the largest value isn't there anymore (i.e. the array is one smaller)



To implement this algorithm we need a few more specifics. If the bubblesort function is passed in the array to sort and a number that we should consider to be the last index in the array, i.e.:

```
void bubblesort(int a[], int endIndex)
```

Then:

1. If endIndex is 0 then there is nothing to do, it is an array with one element, so just return. This is our termination condition.
2. Bubble the largest value in the array up to position endIndex
3. Repeat step 1 by recursively calling bubblesort with the same array a, but send in endIndex -1 instead

Here is code, we will step through it in class!

```
void swap(int &x, int &y)
{
 int temp = x;
 x = y;
 y = temp;
}

void bubblesort(int a[], int endIndex)
{
  if (endIndex == 0)
        return;
  for (int i = 0; i < endIndex; i++)
        if (a[i] > a[i+1])
                swap(a[i],a[i+1]);
  bubblesort(a, endIndex - 1);
}

int main()
{
  int a[] = { 5, 23, 1, 21, 17, 4, 9};
  bubblesort(a, 6);
  for (int x : a)              // Unix compile with -std=c++11
        cout << x << endl;
}
```

There is not really any advantage to the recursive bubble sort over a non-recursive one (in fact the non-recursive has some advantages, can you think of them?) But there are other sorting algorithms that are much easier to write recursively and is the preferred approach.