CMSC131

Recursion Part 1

Iteration Recap

We’ve already seen the power of iteration in computing via the use of **for** and **while** loops.

- **for** loops are common if there is a task that is to be performed a fixed number of times in a basic order/progression of what’s “next” to work on
  - the “**for each**” style provides an easy way to visit everything in (for example) a collection exactly once
- **while** loops are common if you do not know in advance how many times the task needs to be performed or if the order/progression is not based on a simple notion of a “next” thing

Self-Referential Acronyms

There was an e-mail program named ELM.

A group wrote a free e-mail program that was meant to replace it.

They named in PINE.

The acronym meant: **P**INE **I**s **N**early **E**LM

Lazy person with a cloning ability…

Imagine a person who doesn't want to walk more than one step a day but who can clone themselves where they are standing.

They want to get a Pepsi from the refrigerator, but it is many steps away.

What could they do?

Finding the “Minimum” item

How would you describe a loop that would find the minimum item in a list?

Could you describe a way to find it where your description refers to itself?

Describing certain tasks

There are many scenarios where the easiest or most natural way to describe a task might be in terms of smaller versions of the same task.

- To print a list of inputs backwards, print the list from the second element onward backwards, then print the first element.
- To sort a list, split it in half, sort each half, then merge those two sorted lists together.
- To change file permissions for a folder, change the permissions of the files in the current folder and then for every subfolder, change the file permissions in that folder.

Recursion

Most programming languages provide a way to have a subroutine (method in Java) call itself if desired.

- It is important to make sure that you are calling the subroutine on a ***smaller version*** of the same problem since the compiler will not check for this and if it's not a smaller version then the program might never terminate.
- It is also important to have a well-defined ***stopping point*** that does not use a recursive call (sometimes called a base case).
- It is important to make sure you are doing the correct pre and post processing "around" the recursive calls.

Factorial

We define ***n*!** (pronounced **en factorial**) as the result of multiplying all of the numbers in the sequence of descending natural numbers from ***n*** down to **1**.

How could we write a program to compute this value for us?

Factorial: for

The following static method will compute ***n*!**

- factorial grows very fast so the return value has been made **double** to avoid overflow but this can sacrifice some precision.

```
public static double factorialFor(int n) {
double returnValue = 1;
  for (int currVal=n; currVal>0; currVal--) {
    returnValue *= currVal;
  }
  return returnValue;
}
```

Factorial: recursion

We could define factorial recursively as such:

"*n*! is *n* multiplied by (*n*-1)!"

Within this is a certain assumption though that ***n*** starts positive and that when ***n*** gets down to **1** the answer is simply **1**.

Take a minute now and use the above definition to trace through computing 5!

Factorial: recursion

The following static method *also* computes ***n*!** but this time using recursion.

- Again, since factorial grows quickly the return value is **double** to avoid overflow but this can sacrifice some precision.

```
public static double factorialRecur(int n) {
   if (n<=1) return 1; //Stopping point
   return n*factorialRecur(n-1);  //Recursive call
}
```

*Let's draw a representation of **factorialRecur(5)***

How do you feel about the style of the code (lack of {} and a return from the middle)?

- 20% A. Dislike it a lot.
- 20% B. Dislike it a little.
- 20% C. No opinion.
- 20% D. Like it a little.
- 20% E. Like it a lot.

```
public static double factorialRecur(int n)
{
  if (n<=1) return 1;
  return n*factorialRecur(n-1);
}
```

How do you feel about the style of the code for this recursive method?

- 0% A. Dislike it a lot.
- 0% B. Dislike it a little.
- 0% C. No opinion.
- 0% D. Like it a little.
- 0% E. Like it a lot.

```
public static double factorialRecur(int n)
{
  return (n<=1)?1:n*factorialRecur(n-1);
}
```

Factorial discussion

Some things to note and discuss about this problem:

- All of the previous solutions are correct but the use of exceptions might be desired for negatives.
- For large values they will give slightly different results due to the order in which the numbers end up being multiplied and the way floating point math rounding is done.
- While the recursive solution might look more elegant, it has more runtime overhead since each method call has a “cost” to it (we’ve seen some of that with our stack traces).

Potential Problem

In fact, the recursive version of solving factorial ***will not work*** for input over a certain size even if the return value could be stored in a large enough variable.

Why do you think that's the case?

Recursion "hidden" cost

Every time a method is called, a new stack frame is added to the stack to store local variables.

- That stack frame isn't disposed of until that call to the method has completed execution.

This means that every time a method calls itself, a new stack frame is created without the current one being disposed of yet (since it is still going to be needed).

- That's going to be ***a lot*** of memory being used in the case of factorial for a large ***n***.

Scenario: Printing Backwards

To print a list of inputs backwards, we could…

- Use a loop and a data structure such as an **array** or an **ArrayList** to read in and store all of the input, and then another loop and that data structure to print the values out in reverse order.
- We could also express the solution using a recursive definition and say we will print the list from the second element onward backwards, then print the first element.

Print Backwards: **for**

```
public static void printFor(int n) {
int[] values = new int[n];
  for (int index=0; index<n; index++) {
    System.out.print("Enter value: ");
    values[index] = sc.nextInt();
  }

  for (int i=values.length-1; i>=0; i--) {
    System.out.print(values[i] + " ");
  } //i used to make it fit on slide
}
```

Print Backwards: **recursion**

```
public static void printRecur(int n) {
  System.out.print("Enter value: ");
  int val = sc.nextInt();
  if (n==1) {
    System.out.print(val);
  }
  else {
    printRecur(n-1);
    System.out.print(val);
  }
}
```

Pros and Cons

Something a recursive solution might handle more easily/efficiently would be if the number of values were not known in advance (so perhaps a “type -1 to stop” scenario).

Something a for loop solution might handle more easily/efficiently would be if you needed to print an end of line marker at the end of the reversed list.

Are there any other advantages/disadvantages that you see in this case?

Computing Triangular Constructions

- Imagine we have a large supply of 1” cubes. We like to make triangles on the floor in such a way that the top “row” has 1 cube and then the next “row” has 2 cubes, and the next one has 3 cubes…
- Let’s work out how we could write a completely recursive method that uses neither loops nor multiplication to determine the **total** number of cubes that would be used to create a triangle, based on a height passed into the method.
  - We need a stopping/base scenario.
  - We need a recursion definition.

Example Solution

```
public int triangle(int rows) {
   if (rows == 0 ) return 0;
   return rows+triangle(rows-1);
}
```

Fibonacci Numbers

Generating Fibonacci numbers are a common example used when discussing recursion but is also a poor use of it. `Fib(n) = Fib(n-1)+Fib(n-2)`

The execution time a **for-loop** implementation grows in a linear fashion with the size of the problem, but the run time will grow *exponentially* when the "obvious" recursive approach is used.

- CMSC351 will show a technique called memoization that can be used to make the recursive version significantly faster.