Computer Science 225
Advanced Programming
Siena College
Spring 2017

Topic Notes: Recursion

By now, you are all getting pretty good at thinking about problems recursively and turning those recursive algorithms into recursive methods.

Next, we want to think about recursion a bit. Think about the reursive methods you've written and what they have in common and how they're different. And we'll talk about when it makes sense to use recursion and when it's best to use a good ol' loop.

You have certainly noticed that courses in the Computer Science program tend to bring up recursion over and over... You probably started with some mathematical problems (computing factorials, powers, Fibonacci sequences), where it is often easiest to see how a problem can be cast in terms of a smaller instance or subproblem. Then as you were continually asked to consider recursion in more varied situations (*e.g.*, operations on linked lists and tree structures), you were able to see places where recursion is useful.

Hopefully as you have worked through these, you have noticed some of the common characteristics of successful recursion, including that

1. recursion needs to stop
2. the code is *reentrant*

In order for recursion to stop, each call needs to make progress toward a base case.

Reentrant code can have multiple, simultaneous invocations without undesired effect on global variables (*i.e.*, instance variables). You would also need to ensure that values of local variables from one invocation would not need to affect other invocations. It is important to keep in mind that all invocations share a single copy of instance or class variables, and each invocation has its own copy of parameters and local variables. Not to say that recursive calls should never modify instance variables, but care must be taken.

One fact to keep in mind is that you never *need* recursion to solve any problem. Every recursive program can be rewritten iteratively. At the worst, you can always simulate the recursion by creating and managing your own stack to play the role of the call stack that's keeping track of your recursive calls and their parameters and local variables.

So when should we use recursion? Recursion comes with a price (all those method calls and the stack to keep track of their parameters, local variables, and return information.

If a recursive process can just as easily be written as a simple loop, then (unless your instructor forces you to use recursion) don't use recursion! For example, we could find the largest value in an array of integers recursively:

```
  public static int max(int[] a) {

    return max(a, 0);
  }

  private static int max(int[] a, int startAt) {

    if (startAt == a.length-1) {
      return a[a.length-1];
    }
    int maxOfRest = max(a, startAt+1);
    return (maxOfRest > a[startAt] ? maxOfRest : a[startAt]);
  }
```

This is overly complex and is less efficient than a simple loop version:

```
  public static int max(int[] a) {
    int maxVal = a[0];
    for (int i = 1; i < a.length; i++) {
       if (a[i] > maxVal) maxVal = a[i];
    }
    return maxVal;
  }
```

You might call that “dumb” recursion.

There’s also careless recursion. Consider three ways to compute a value to a power:

```
    // compute the power using a good old fashioned loop
    public static int loopPower(int base, int exponent) {

        int answer = 1;
        for (int i=0; i<exponent; i++) {
            answer *= base;
        }
        return answer;
    }

    // the straightforward recursive approach
    public static int recPower(int base, int exponent) {

        // our base case is exponent == 0
        if (exponent == 0) {
            return 1;
        }

        // otherwise, we have to do some work, b^n = b * b^{n-1}
        return base * recPower(base, exponent -1);
    }

    // a more efficient recursive approach, based on the idea
    // that we can compute a power b^{2n} as (b*b)^n
    public static int fastRecPower(int base, int exponent) {

        // base case is again exponent == 0
        if (exponent == 0) {
            return 1;
        }

        // now, see if the exponent is even or odd
        if (exponent % 2 == 1) {
            // it's odd, so use straightforward recursion to get
            // down to an even case
            return base * fastRecPower(base, exponent - 1);
        }

        // if we got here, it's even, so we can do better
        return fastRecPower(base * base, exponent / 2);
    }
}
```

The last one is good, but suppose instead we replaced the last line with the seemingly equivalent:

```
        return fastRecPower(base, exponent / 2) *
               fastRecPower(base, exponent / 2);
```

This will make two identical calls in a row, making the code much less efficient. Instead, we could do:

```
        int oneCall = fastRecPower(base, exponent / 2);
        return oneCall * oneCall;
```

Or worse yet, consider the naive recursive method for computing a value in the Fibonacci sequence:

```
public static long fib(int x) {

    if (x < 2) return 1;
    return fib(x-1) + fib(x-2);
}
```

Even though we have 2 different recursive calls, there is a huge amount of redundant computation.

Here we really don't want recursion at all. It's tempting to write this with an array:

```
public static long fib(int x) {
    long vals[] = new long[x];
    vals[0] = 0;
    vals[1] = 1;
    for (int i = 2; i < x; i++) {
       vals[i] = vals[i-1] + vals[i-2];
    }
    return vals[x-1] + vals[x-2];
}
```

But even this is unneccesarily complicated. We don't need that whole array, we just need the last two values.

```
public static long fib(int x) {
    long fibiminus2, fibiminus1, fibi;
    fibiminus2 = 0;
    fibiminus1 = 1;
    for (int i = 2; i <= x; i++) {
       fibi = fibiminus1 + fibiminus2;
       fibiminus2 = fibiminus1;
       fibiminus1 = fibi;
    }
    return fibi;
}
```

Now we only need to remember 3 values no matter how large x is.

In many cases, recursion is very appropriate and makes for an elegant way to approach a problem.

You have almost definitely seen many recursive methods on tree structures. A recursive data structure naturally lends itself to recursive methods.

A classic example you've probably seen before is the Towers of Hanoi problem.

See Example: Hanoi

Another example of a problem naturally solved by recursion is bin packing.

See Example: BinPacking