SIENA*college*
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2019

Topic Notes: Recursive Methods

You have seen in this course and in your previous work that iteration is a fundamental building block that we can use to construct code that implements an algorithm to solve a problem. We will next spend some time considering another fundamental building block: *recursion*.

The idea here is that we solve a problem by first solving one or more smaller instances of the problem, then using those solutions, to solve the original problem.

As a simple example suppose I want to add up everyone's midterm scores in a class of n students. A recursive approach to this would be to add up the $n - 1$ scores from all but one person, giving me a solution to a smaller instance of the same problem (a problem of size $n-1$ instead of n), then taking that total and adding in the score of the person whose exam was initially omitted, giving me the solution to the original problem.

The recursive approach would solve each of the subproblems recursively as well. That is, the $n-1$ exams would be tallied by first computing the total of $n - 2$ of them and adding in the $n - 1^{\text{st}}$. If you think about this a bit, you'll also realize we need some way to stop the recursion. Here, we have two choices, either of which would work. We could say that to add up the exam scores for a set of 1 exam, we just take that exam's score as the answer. Or, we can say that to add up the exam scores for a set of 0 exams results in a total score of 0.

So we can write this procedure a bit more precisely as a set of instructions:

```
To add scores on n exams:
  if there is just one score, the answer is that exam's score
  else
    compute the scores of all but 1 exam using these instructions
    add in the exam that was not part of the subset computed
    return that sum as the answer
```

Recursive Methods

For our first coding example with recursion, we first look back at an older example:

See Example: Sum1ToN

In particular, our method to compute the sum:

```java
public static int sumNumbersTo(int limit) {
    int sum = 0;
```

```
for (int number = 1; number <= limit; number++) {
    sum += number;
}
return sum;
}
```

For the moment, forget this formula:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

and the fact that it led us to a much more efficient way to compute the sum. We'll focus on a formula that represents the iterative method above:

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n$$

The "..." in the formula is encoded in Java by the `for` loop.

Alternatively, we can think about this formula recursively:

$$\sum_{i=1}^{n} i = \begin{cases} 1 & \text{if } n = 1 \\ (\sum_{i=1}^{n-1} i) + n & \text{otherwise} \end{cases}$$

This says that we can compute the sum of the first n numbers by applying the appropriate rule:

- If $n = 1$, the sum is trivial and we know the answer is 1. This is called the *base case* for our recursion.

- Otherwise, we will apply this same formula to compute the sum of the first $n - 1$ numbers, and then add n to that sum to get our answer. This is the *recursive case*.

In the end, we'll need the same number of addition operations as we did in the iterative formula.

So let's look at this translated to a Java method:

See Example: Sum1ToNRec

```
public static int sumNumbersTo(int limit) {

    // first, we check for the base case
    if (limit == 1) return 1;

    // otherwise, we have to make a recursive method
```

```
        // call to compute the sum of the first limit-1
        //  numbers, then add in limit to get our answer
        return sumNumbersTo(limit - 1) + limit;
    }
```

The above method looks very simple, but it often completely confuses programmers who are not experienced in working with recursion. Yes, the method **calls itself** to be able to compute its answer.

How can this work? It's the equivalent of you going up to an expert on addition and asking how to add together a bunch of numbers. The (unsatisfying) response is "add a bunch of numbers then add one more time". If you knew how to add a bunch of numbers, you wouldn't be asking in the first place (so you ask again)! It's like looking up a word in the dictionary only to have the dictionary use the word in its definition.

But the key here is that each time you ask, there are fewer numbers to add up. Eventually, you will ask the expert on addition to add a single number, to which the response will finally be "oh, you have just one number there, the sum is that number."

The same thing happens here. Your program keeps calling sumNumbersTo with smaller and smaller values for the limit parameter until limit finally becomes 1, and that method returns 1. The big question is, how do we know how many times it took to get to that point? They key idea is that your program now has n copies of the limit parameter, whose values range from n on the initial call, down to 1 on the call that triggers the base case.

I think that to understand this, we need to construct a memory diagram (which will be done in class).

Comparing Costs

Another thing we want to start to think more carefully about is the relative costs of different ways to perform a computation. There are many measures of cost that can be interesting in different contexts, but the most common are *computational cost*, the amount of work the processor needs to perform to complete the task, and *memory cost*, the amount of memory that needs to be allocated to the program to complete the task.

These values are typically measured as a function of some input parameter or size.

We have three versions of the sumNumbersTo method that we can consider in this context: the iterative version, the recursive version, and the one that uses the one-step formula. In these methods, we will think of the "problem size" as the input number – the number of values we want to add up. Subsequent discussion here will refer to this as n, even though it is named limit in the formal parameter of the methods.

First, we consider the amount of computing. Here, a meaningful measure is how many arithmetic operations are required.

For the iterative method, the addition operations are inside the loop – one per iteration. The loop will execute $n - 1$ times, so there is a total of $n - 1$ additions required.

For the method that uses the formula, there is a total of 3 operations, one each of addition, multiplication, and division. Note that this value stays the same no matter what n is. This is good, especially for large values of n.

For the recursive method, we need to think a little more. Each time the recursive case executes, there are two operations in play – we have to do one addition and one subtraction (to compute `limit-1`). This might at first seem more expensive than the iterative version, but there we ignored the n additions needed to manage the `for` loop. There are no arithmetic operations needed when the base case executes. So what remains is to determine how many times the recursive case executes before we reach the base case. That number is $n - 1$ here.

In the first and last cases, we note that the computational cost is directly proportional to n – that is, the computational cost scales linearly as n increases. A more formal way of saying this is that the computational cost of these is $O(n)$, usually read "order n" or "Big-O of n". In the method that uses the formula, the cost is a constant. That is, it stays the same no matter how large a value of n we present. More formally, this is a method that executes in $O(1)$ time.

The memory costs are measured by detemining how many parameters and local variables (often called *stack variables*) as well as objects and/or arrays constructed with `new` (called *heap variables*) are allocated. In these methods, there are no heap variables involved, so we can focus on the formal parameters and local variables.

In the non-recursive methods, the method executes just once to complete its work, so there is just the one copy of each parameter and local variable. The iterative version has three such variables: the parameter, the local variable `sum`, and the loop index variable. The formula version has just the parameter. In either case, the amount of memory needed does not depend on n, so the memory usage is constant, or $O(1)$.

The recursive method's memory cost does depend on n, as each recursive call results in another copy of the parameter `limit` being created on the stack. Note in particular how each method call is still in execution until the chain of recursive method calls reaches the base case. We said earlier that there will be a total of $n - 1$ recursive calls, and at the peak depth of the recursion, all $n - 1$ are in execution. This means the amount of memory needed is linearly proportional to n, or $O(n)$.

A Slightly More Interesting Example

As a next example, let's look at a program to raise numbers to powers:

See Example: Powers

We can see that the basic idea here is to read in a couple of integers, a base and an exponent, and then raise the base to that power. There are three methods here, all of which compute the same thing but in different ways.

Before we look at the three methods and the `main` method that uses them, first a few words (reminders, for many of you) about the ranges of values that can be represented by Java's integer types. Normally, we use the `int` data type, which uses a 32-bit 2's complement format to represent numbers between -2^{31} and $2^{31} - 1$. This is sufficient for the vast majority of programs. If that's not enough, we can move up to the `long` primitive type, which uses 64 bits and can represent values

between -2^{63} and $2^{63} - 1$. You will see that the program uses `long` values for the exponents, and also uses Java's long integer literal format by placing an "L" at the end of the integer literals. `long` variables can hold some really big numbers, but we can easily exceed those limits when computing powers. So this program uses a Java API class called `BigInteger` to represent base values and computed products, which allows for arbitrarily-large integer values to be represented precisely.

Now on to the methods.

The first, `loopPower` simply contains a `for` loop to multiply `a` by itself `n` times. Correct, but not especially interesting.

Mathematically, we are computing

$$a^n = 1 \cdot \underbrace{a \cdot a \cdot ... \cdot a}_{n \text{ times}}$$

We will focus on the others, first `recPower`. How does this one work? Clearly, if we evaluate `recPower(3,0)`, the condition `n == 0` is `true`, so the method should return 1. Suppose instead we evaluate `recPower(3,1)`. According to the method definition and the fact that `1!=0`, we get that `recPower(3,1) = 3*recPower(3,0)`, and we know the value of `recPower(3,0)` is 1. Thus the final answer is `3*1` or 3.

Mathematically, we are using this *recurrence*:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a \cdot a^{n-1} & \text{if } n > 0 \end{cases}$$

Because we are calculating complex powers using simpler powers (the mathematical recurrence is on a problem with a smaller exponent, and in the method, recursive calls each are passed an exponent one smaller than the one with which this call was passed), we eventually get to our base case.

It sometimes helps to imagine that we are having someone else handle the recursive call. That is, if I want to calculate `recPower(3,5)`, I ask someone else to calculate `recPower(3,4)`, without caring how they do it, and then, when they give me the answer, `81`, multiply that answer by 3 to get the final answer of `243`. It just happens that that "someone else" is using the same method we're writing!

It is often helpful to use a recursive call diagram, a bit like a simplified memory diagram, to help understand what is happening with recursive methods. For example, a call to `recPower(2, 8)` would result in a diagram like this:
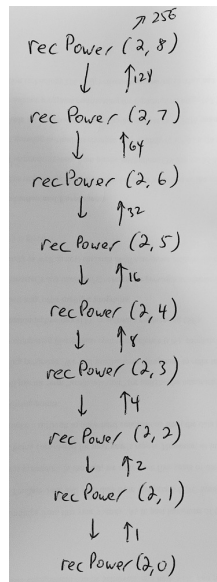
Using a simple modification of the above recursive method we can get a very efficient algorithm for calculating powers, shown in `fastRecPower`. In particular, if we use either of the first two methods, it will take 1024 multiplications to calculate 3^{1024}. Using a slightly more clever algorithm we can cut this down to only 11 multiplications!

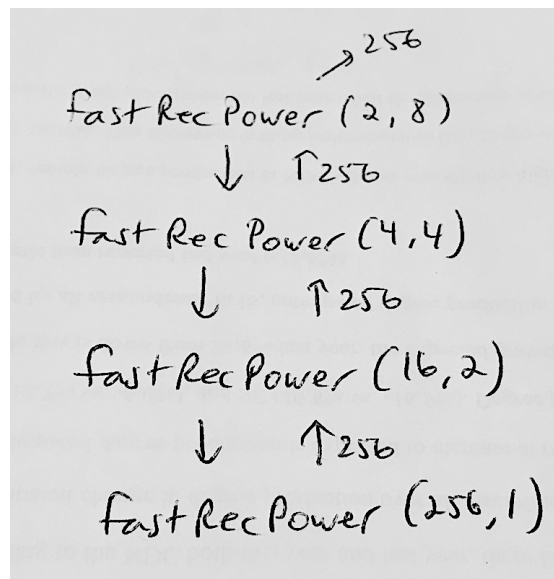The mathematical recurrence for our improved method is:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a \cdot a^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (a \cdot a)^{\frac{n}{2}} & \text{if } n > 1 \text{ and } n \text{ is even} \end{cases}$$

In each of the first two methods, the number of multiplications necessary is equal to the value of the exponent. That is not the case here.

```
fastRecPower(3,16) = fastRecPower(9,8)    // mult 3*3
    = fastRecPower(81,4)                  // mult 9*9
    = fastRecPower(6561,2)                // mult 81*81
    = fastRecPower(43046721,1)            // mult 6561*6561
    = 43046721 * fastRecPower(43046721,0)
    = 43046721 * 1                        // mult
    = 43046721
```

Thus it only took 5 multiplications (and 4 divisions by 2) using `fastRecPower`, whereas it would have taken 16 multiplications the other way (and divisions by two can be done very efficiently in binary).

For the example for which we drew a recursive call diagram, that becomes:

$$\text{fastRecPower}(2,8) \nearrow 256$$
$$\downarrow \quad \uparrow 256$$
$$\text{fastRecPower}(4,4)$$
$$\downarrow \quad \uparrow 256$$
$$\text{fastRecPower}(16,2)$$
$$\downarrow \quad \uparrow 256$$
$$\text{fastRecPower}(256,1)$$

In general it takes somewhere between $\log_2 n$ and $2*\log_2 n$ multiplications to compute a power this way. While this doesn't make much of a difference for small values of n, it does make a significant difference when n is large. For example, computing `fastRecPower(3,1024)` would only take 11 multiplications, while computing it with either of the other two methods would take 1024 multiplications.

The key is that by rearranging the order of doing things in a clever way, we can cut down the amount of work considerably! (Again it is possible to write the above algorithm with a `while` loop, but the above recursive formulation is arguably easier to understand.)

Let's think a bit more carefully about the relative costs, in terms of both computation and memory.

The costs of the iterative `loopPower` and straightforward `recPower` methods are just like the iterative and recursive methods we saw above. Recall that n represents the exponent to which we are raising a number.

The iterative method has a loop that executes n times for a computational cost of $O(n)$. The memory cost is constant with just a single method call with a couple of parameters and local variables allocated, so the memory cost is $O(1)$.

The recursive method executes its recursive case n times, with a constant amount of work on each of those executions, for a total computational cost of $O(n)$. However, each of the method invocations results in another copy of the parameters being created. So memory cost is $O(n)$.

The `fastRecPower` method is much more interesting. Each call to the method executes one of the three cases, each of which itself results in only a constant amount of computation and one copy of the parameters on the stack. We mentioned about that it takes somewhere between $\log_2(n)$ and $2*\log_2(n)$ recursive steps, so the computational and memory costs are both $O(\log n)$.

Recursion with Arrays

Recursive methods can also be used with arrays. For example, suppose we want to find the largest element in an array of `int`. We can write a method quite easily to do this with a loop:

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

However, the problem can also be decomposed in a recursive manner by thinking about subarrays:

- The largest value in a subarray of size 1 is the value of that 1 element in the subarray

- The largest value in a subarray with at least 2 elements is the larger of the value in the first element and the largest in the subarray consisting of all but that first element

Or as a method, where we pass in the first element of the subarray we wish to consider:

```
public static int maxRec(int[] a, int start) {
  // base case: looking at the last element
  if (start == a.length-1) return a[start];

  // recursive case: max of a[start] and the
  // max of the rest
  int maxofRest = maxRec(a, start+1);
  if (maxOfRest > a[start]) return maxOfRest;
  return a[start];
}
```

Such a method is often combined with a helper method that tacks on the extra parameter that we would not want a user of our method to have to include (as it would normally be 0 to start the search for the max at position 0):

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

   return maxRec(a, 0);
}
```

Similar methods can be used on `Strings`, which are after all, essentially arrays of characters.

Recursive Method Summary

We can both write and understand recursive programs as follows:

1. Write the base case. Convince yourself that this works correctly.

2. Write the "recursive" case.

   - Make sure all recursive calls go to simpler cases than the one you are writing. Make sure that the simpler cases will eventually get to a base case.

   - Make sure that the general case will work properly if all of the recursive calls work properly.