Topic Notes: Recursion and Mathematical Induction

Recursion

An important tool when trying to solve a problem is the ability to break the problem down into some number of smaller sub-problems, the solutions to which you can use to solve the original problem.

Oftentimes, those sub-problems look a lot like the original problem. In fact, they might be the *same* problem, just on a smaller set of input data.

These kinds of problems often have a *self-referential* or *recursive* solution.

It's a strange idea at first – calling the method you're writing before you're done writing it. Well, if my method needs to call my method to finish up, how am I ever going to get anywhere?

Many algorithms may be recursive. Once you are used to them, they can be easier to understand (and prove correct) than iterative algorithms.

Even if you're fairly comfortable with recursion, I expect that few of you have formally proven properties about recursive algorithms before. We will be doing some of that soon.

We start with a simple and classically recursive example: computing a factorial.

$$n! = n \cdot (n-1) \cdot ... \cdot 1$$

We could write a simple method to compute this with a for or while loop. But it is just begging to be solved recursively.

$n!$ is nothing more than $n \cdot (n-1)!$. So to compute it, all we do is compute $(n-1)!$ (which is certainly easier than computing $n!$) and then multiply by n and we have the answer.

```
public static int factorial(int n) {

  return n*factorial(n-1);
}
```

That's great. We were writing a method to compute factorials anyway, so why not call it? Assuming we know how to compute $(n-1)!$, we now can compute $n!$.

Problem: what about 2? $2! = 2 * (1!) = 2 * 1 * (0!) = 2 * 1 * 0 * (-1!)$... That won't work. We need to stop the recursion somehow.

We need a *base case*. Well, 1!=1, so let's stop our factorial when it gets to 1:

```
public static int factorial(int n) {

  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

The keys to a successful recursive solution: identify the base case and make sure the recursive step is making *progress* toward the solution (closer to the base case).

Pros and Cons of Recursion

- Disadvantages
  - Any recursive program requires additional (implicit) storage on the run-time stack.
  - There is also a slight run-time overhead in the procedure calls since the system must push activation records onto the stack.
  - Execution is often slightly faster and will use less space if it can be done iteratively.
- Advantages
  - Often it is easier to construct a recursive solution.
  - The resulting code may be significantly more clear.
  - Smart compilers (particularly for functional languages) will remove "tail" recursion automatically.

For our purposes, we will often begin with a recursive approach where appropriate. However, we do want to keep in mind the extra costs associated with this approach. In cases where efficiency is important, we may wish to eliminate or at least limit recursion. The good news: it's always possible to eliminate recursion. The bad news: it often complicates the code.

Example: Making Change in Postage Stamps

Consider the example from the text: making change in postage stamps.

This is more interesting than making change with U.S. currency. A *greedy* approach works well with currency. We will always use the largest denomination that is less than or equal in value to the amount we still need to account for. With postage stamps, if we want to receive our change using the fewest number of stamps, it's not that easy.

As of September 2009, the "useful" stamps in which to receive change would be 44 cent letter and 28 cent postcard, with any smaller amounts made up in 1 cent penny stamps. Thus, the greedy approach fails. For example, if we were to require 57 cents in change, the greedy approach would result in 1 letter stamp and 13 penny stamps, for a total of 14 stamps, while a far better solution would consist of 2 postcard and 1 penny stamp, for a total of 3.

Here is a straightforward solution:

See Text Example:
`/home/jteresco/shared/cs211/eg/structure5/RecursivePostage.java`

This works, but it is pretty inefficient. The problem is that lots of subproblems are solved over and over each time the result is needed. The complexity of this solution is exponential ($O(3^n)$), so we would like to be able to do better to solve large instances of the problem in a reasonable amount of effort.

We can improve upon this by trading some space to save time. If we *cache* (remember) the results of the subproblems we compute along the way and reuse them when possible, we can greatly reduce the number of recursive calls. This is a *dynamic programming* approach, and will result in a linear ($O(n)$) running time, at the expense of linear space.

Mathematical Induction

One of our goals in our study of data structures and algorithms will be to prove (formally) the correctness and/or complexity of the structures and algorithms.

We will make use of a proof technique called *mathematical induction*.

Formally, the *principle of mathematical induction* can be stated:

Let A be a set of natural numbers such that

1. 0 is an element of A, and
2. for each n, if $0, 1, ..., n$ in A, then $(n + 1)$ is also in A. Then A is the set of natural numbers.

We use this to prove statements of the form:

For all natural numbers n, P is true.

A proof by mathematical induction typically follows a 5-step template.

1. State your intend to use induction: "We proceed using mathematical induction on the size of the problem" or some similar statement.
2. Prove the *base case* or base cases. This is often very straightforward and involves a trivial or at least simple case or cases.
3. State your the assumption that the statement you are attempting to prove holds for all values from the base case up to but not including the nth case. This is your *inductive hypothesis*. (You may only need to assume the $n - 1$st in many cases, so do that if possible.)
4. Prove, using the assumption that the simpler cases hold, that the nth case holds. This is the *inductive step*.

5. Claim that the statement is true for all n by mathematical induction.

A Simple Example

For example, we can prove by induction that

$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

Proof: We proceed by mathematical induction on n.

Our base case is $n = 0$. The formula in this case yields $\frac{0(0+1)}{2} = 0$, which is correct.

Now, we assume that the formula is correct for all values between 0 and $n - 1$.

For n, the sum on the left-hand side may be written:

$$0 + 1 + 2 + ...(n - 1) + n$$

we can rewrite this

$$[0 + 1 + 2 + ...(n - 1)] + n$$

The part in brackets is, by our inductive hypothesis, $\frac{(n-1)n}{2}$. Substituting this into our expression above yields:

$$\left[\frac{(n-1)n}{2}\right] + n$$

We can simplify this:

$$\frac{(n-1)n}{2} + \frac{2n}{2} = \frac{(n-1)n + 2n}{2} = \frac{n(n+1)}{2}$$

Therefore by mathematical induction, we have shown that the formula holds for all n. $\diamond$

Another Example

Prove that

$0^3 + 1^3 + 2^3 + ... + n^3 = (0 + 1 + 2 + ... + n)^2$

Proof: We proceed by mathematical induction on n.

Base: $n = 0$. $0^3 = 0 = (0)^2$ [Done.]

Inductive hypothesis: Assume that our formula holds for values up to $n - 1$.

Inductive step: We show, using this assumption, that the formula holds for n.

$$
\begin{aligned}
0^3 + 1^3 + 2^3 + ... + n^3 &= 0^3 + 1^3 + 2^3 + ... + (n-1)^3 + n^3 \\
&= (0 + 1 + 2 + ... + (n-1))^2 + n^3 \\
&= \left(\tfrac{n(n-1)}{2}\right)^2 + n^3 \\
&= \left(\tfrac{n^2-n}{2}\right)^2 + n^3 \\
&= \tfrac{n^4-2n^3+n^2}{4} + \tfrac{4n^3}{4} \\
&= \tfrac{n^4+2n^3+n^2}{4} \\
&= \tfrac{(n^2+n)^2}{4} \\
&= \left(\tfrac{n(n+1)}{2}\right)^2 \\
&= (0 + 1 + 2 + ... + n)^2
\end{aligned}
\tag{1}
$$

By mathematical induction, this formula holds for all n. $\diamond$

A Bad Proof

We do need to be careful with mathematical induction, however. Consider the following "proof".

Prove: All cows are the same color.

Proof: Proceed by mathematical induction.

Base case: consider a group of one cow. Clearly, all cows in the base case are the same color.

Inductive hypothesis: Suppose that for any group of fewer than n cows, they are all the same color.

Inductive step: Prove that a group of n cows is all of the same color. So let's take out one cow from the group of n. Our induction hypothesis states that the remaining group of $n-1$ cows are all of the same color. Put that cow back and take out a different cow, leaving another group of $n-1$ horses, which, by our inductive hypothesis, are all the same color. Therefore, the entire group of n is of the same color.

Why does this fail? It's the two-horse case that's the problem. There are no horses in common in the "other" group, so we can't say anything about the horses being the same color.

For another example of a bad "proof", see also the proof that all Canadians are the same age, linked from the lecture.

Proving algorithm correctness and time complexity will come later.