Lecture Notes 5 – Recursion and induction
CSS 501 – Data Structures and Object-Oriented Programming

Reading for this lecture: Carrano, Chapter 5

To be covered in this lecture:
- More recursion
- Induction

Once again:
- **Passing by Value:**
```
void MyFunc(const MyDataType& passByValue);
```
- **Passing by Reference:**
```
void MyFunc(MyDataType &passByRef);
```
- **Left over useless "feature" from C-days:**
```
void MyFunc(MyDataType doNotDoThis);
```

Writing recursive methods
Let's review briefly some strategies for writing recursive methods.
1. Write down precisely what your function does. You must believe that it will do this task, even though you haven't written it yet.
2. Solve the base case (which is usually easy). For example: n = 0, an empty array, an array with one element, no operations to perform..
3. Solve the recursive part logically. Take your problem and break it down into parts, where one part is basically the same problem, but smaller than the original problem. Examples:
    a. A smaller array (one element smaller or half as big)
    b. A smaller number of operations – computing (n-1)! or n – 1 disks to move
4. Code the recursive part. Be sure to think in terms of step #1 and call your function if you find that you need to solve that task. It takes some getting used to because you are using what you are writing. Pretend it already exists, that you are just calling a function as you normally do.

The Fibonacci sequence
There is a famous sequence of numbers called the Fibonacci sequence. The textbook describes it in terms of breeding rabbits using the following assumptions:
- Rabbits never die.
- A rabbit is able to reproduce when it is two months old.
- Rabbits are always born in male-female pairs.
- Each month, every pair of rabbits that is able to reproduce gives birth to one male-female pair.

Starting with a single pair of newborn rabbits, how many rabbits are there after 3 months? 6 months? n months?

Month 1: 1 pair
Month 2: 1 pair (not old enough to reproduce)
Month 3: 2 pairs (the first pair reproduced)
Month 4: 3 pairs (the first pair reproduced, but the new pair was not old enough)
Month 5: 5 pairs
Month 6: 8 pairs
Month n: number at month n – 1 plus number at month n –2

The Fibonacci sequence is usually defined as $F(n) = F(n – 1) + F(n – 2)$, where $F(0) = 0$ and $F(1) = 1$. (Definitions sometimes vary in whether the first term is 0 or 1, but this only changes the indexes by one.)

We can compute the number of rabbits at any month using a recursive function:

```
// Computes the Fibonacci sequence
int fib(int n) {
        if (n < 1) return 0;
        if (n == 1) return 1;
        return fib(n – 1) + fib(n – 2);
}
```

This procedure uses two recursive calls unless the base case is reached. This means that a trace of the procedure calls for an example like **fib(6)** will branch like a tree. This is not an efficient solution for this problem. An iterative solution can solve it much faster for large numbers. This is an example of a problem for which a recursive solution is the easiest to write, but it is not the best solution. It is possible to write an efficient recursive solution to this problem. The key is that it has only one recursive call.

Trace of fib(6):
fib(6) =
(fib(5) + fib(4)) =
((fib(4) + fib(3)) + (fib(3) + fib(2))) =
(((fib(3) + fib(2)) + (fib(2) + fib(1))) + ((fib(2) + fib(1)) + 1)) =
((((fib(2) + fib(1)) + 1) + (1 + 1)) + ((1 + 1) + 1)) =
((((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1)) =
8

Fibonacci sequence revisited

The easy method of writing the Fibonacci sequence is not efficient, since we need to make two recursive calls each time through the function. We can write an efficient recursive function if we count up, instead of down:

```
int fib(int n, int result = 1, int prev = 0)
{
if (n<=0) return prev;
if (n==1) return result;
else return fib(n – 1, prev + result, result);
}
```

Here's what a trace of fib(6) looks like:

```
fib(6) =
fib(5, 1, 1) =
fib(4, 2, 1) =
fib(3, 3, 2) =
fib(2, 5, 3) =
fib(1, 8, 5) =
8
```

There is actually a closed-form solution for the Fibonacci sequence:

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Greatest common divisors and the Euclidean algorithm (optional)

We can write a very simple algorithm to find the greatest common divisor of the two integers as follows:

```
int gcd(int a, int b) {
  a = a > 0 ? a : -a;
  b = b > 0 ? b: -b;
  int smallest = a < b ? a : b;

  for (int divisor = smallest; divisor >= 2; divisor--)
    if (a % divisor == 0 && b % divisor == 0)
```

```
        return divisor;

    return 1;
}
```

Note that the loop in this function will execute **smallest-1** times if the gcd is 1. This means that the worst-case running time of the algorithm is proportional to the smaller of the numbers.

The Euclidean algorithm can do better than this for most cases. Here is an implementation of the Euclidean algorithm:

```
// Preconditions: a > b, a > 0, b >= 0
int gcdEuclidean(int a, int b)
{
        if (b == 0) return a;
        return gcdEuclidean2(b, a % b);
}
```

The analysis of this very simple algorithm is quite interesting (and complex given the simplicity of the algorithm). It turns out that the smallest numbers that require n iterations of this algorithm are F(n-1) and F(n) (consecutive numbers in the Fibonacci sequence). Since the Fibonacci sequence grows exponentially, this implies that the number of iterations of the algorithm is (at worst) logarithmic in the size of the smaller of a and b.

Determining the size of blobs in images

Let's say that an image has been processed such that areas of interest (perhaps abnormalities) have been coded with the red byte equal to 255 and other areas with 0. If we want to determine the size of a blob (connected using neighbors above, below, left, and right) of pixels that are set to 255 starting from some spot in the image, we can use the following recursion. For simplicity, let's use my image struct, although you shouldn't be doing this in your drivers anymore.

```
// Precondition: every red byte is set to 0 or 255
// Postcondition: the size of the blob containing startRow,startCol is returned
int blobsize(image &input, int startRow, int startCol) {

        if (startRow < 0 || startRow >= input.rows) {          // a base case
                return 0;
        }

        if (startCol < 0 || startCol >= input.cols) {
                return 0;
        }

        if (input.pixels[startRow][startCol].red == 0) ||
          (input.pixels[startRow][startCol].red == 1) {       // another base case/or already counted
                return 0;
        }

        input.pixels[startRow][startCol].red = 1;              // mark as counted
        int total = 1 +     blobsize(input, startRow + 1, startCol) +
                            blobsize(input, startRow – 1, startCol) +
                            blobsize(input, startRow, startCol + 1) +
                            blobsize(input, startRow, startCol – 1);
        input.pixels[startRow][startCol].red = 255            // put back to original value

        return total;
}
```

Induction

Induction is a mathematical proof technique that is similar in nature to recursive programming. Consider the hypothesis: 2 + 4 + 6 + 8 + … + 2n = n (n + 1) for all n > 0. Let's look at the first few cases:

n = 1: Left side: 2
Right side: 1 (1 + 1) = 2
n = 2: Left side: 2 + 4 = 6
Right side: 2 (2 + 1) = 6
n = 3: Left side: 2 + 4 + 6 = 12
Right side: 3 (3 + 1) = 12
n = 4: Left side: 2 + 4 + 6 + 8 = 20
Right side: 4 (4 + 1) = 20

The hypothesis is true for the first four cases. Is it true for all cases with n > 0? You can't prove that a statement like this is true by example, but you can prove it false if you find a counter-example. One method to prove statements like this is to use mathematical induction.

The principle of mathematical induction

The principle of mathematical induction is an axiom of mathematics used to prove hypotheses that state something is true for all integers greater than zero (or some other number). A simple form can be stated as follows:

Suppose we have a statement S (n), that is either true or false for all integers, n > 0.
The statement is true for all n > 0, if:

1. **The statement S(1) is true.**
2. **Assuming S(n) is true for every 0 < n <= k, then S(k + 1) is true.**

The first requirement is called the *basis step* or *base case*. The second requirement is the *inductive step*, composed of the *inductive hypothesis* and the *inductive conclusion.* Let's use this to prove our example above. The base case (n = 1) is true as demonstrated above in the introduction. Now, we **assume that 2 + 4 + 6 + 8 + … + 2n = n (n + 1)** is true for **0 < n <= k**. We need to prove that the statement is true for n = k + 1, that is:

2 + 4 + 6 + 8 + … + 2k + 2(k + 1) = (k + 1) (k + 1 + 1) = (k + 1)(k + 2) = k*k + 3k + 2

Since we are assuming that the statement is true for n = k, we have:

2 + 4 + 6 + 8 + … + 2k = k (k + 1)

Substituting this into the inductive conclusion, we get:

k(k + 1) + 2(k + 1) = k*k + 3k + 2
k*k + 3k + 2 = k*k + 3k +2

This proves the inductive step and, thus, the statement must be true for all n > 0.

Intuition

Induction looks like black magic. Are we assuming that something is true to prove that it is true? No.
Think of it this way. We want to prove S(n) is true for n > 0.
First, we prove S(1) is true.
If we prove S(1) implies S(2), then S(2) is true.
If we prove S(2) implies S(3), then S(3) is true.
And so on. The inductive step proves all of statements of this type simultaneously:
If we prove S(0) is true and S(k) implies S(k+1) for every k > 1, then every S(k+1) is true (no matter what k is)!
Thus, all S(n) are true for n > 0.

More examples

Prove that $\sum_{i=1}^{n} 2i - 1 = n^2$ for n > 0

Basis step (n = 1):
LHS = 2(1) – 1 = 1
RHS = 1^2 = 1

Inductive step:
Assume that the statement is true for 0 < n <= k. In particular, we assume that it is true for n = k:

$$\sum_{i=1}^{k} 2i-1 = k^2$$

We need to prove that it is true for n = k + 1.

$$\sum_{i=1}^{k+1} 2i-1 = (k+1)^2$$

We can manipulate the right side to:

$$\sum_{i=1}^{k+1} 2i-1 = 2(k+1)-1+\sum_{i=1}^{k} 2i-1$$

Now, using the inductive hypothesis, we get:

$$2(k+1)-1+\sum_{i=1}^{k} 2i-1 = 2k+1+k^2 = (k+1)^2$$

Prove $3^{2n} - 1$ is divisible by 8 for n > 0.

Basis step (n = 1):
$3^{2\cdot 1} - 1 = 8$, which is divisible by 8.

Inductive step:
Assume that the statement is true for 0 < n <= k. In particular, we assume that it is true for n = k, that is $3^{2k} - 1$is divisible by 8.
We need to prove that the statement is true for n = k + 1.

$3^{2(k+1)} - 1 = 3^2 3^{2k} - 1 = 9\cdot 3^{2k} - 1 = 9\cdot(3^{2k} - 1) + 8$

In the final term above, we have the sum of two terms. Using the inductive hypothesis, the first term must be divisible by 8. The second term is also divisible by 8. Since the sum of any two numbers divisible by 8 is also divisible by 8, the statement must be true.

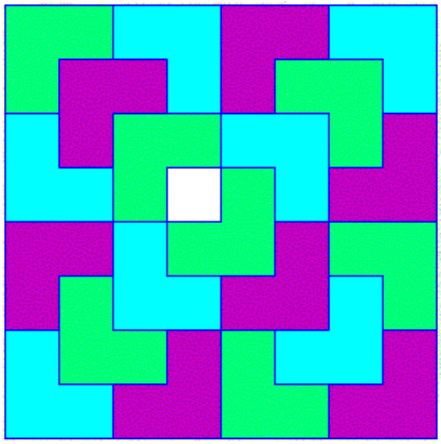
Tiling problem

Now let's examine a completely different type of problem. This is a problem with triominoes. A triomino is a block like a domino, except it has three squares at a right angle (not all in a row). The problem is: Given a board of size 2^n x 2^n ($n > 0$), with exactly one block missing (it could be any block), can the board be tiled with triominoes so that every block (except the missing one) is covered by exactly one triomino and each triomino is completely within the board. Interestingly, the answer is yes and we will prove it.

The base case is n = 1. In this case, the board with one block missing always fits just one triomino and this tiles all of the remaining blocks.

For the inductive case, we will assume that we can tile a 2^k x 2^k board with one block missing. We need to show that we can tile a 2^{k+1} x 2^{k+1} board with one block missing. Solution: Place a triomino in the center of the board such that it has one block in each of the three quadrants that don't contain the missing block. The board can now be divided into four smaller boards, each of which is a 2^k x 2^k board with one block missing. We assumed that we knew how to tile these, so we must be able to tile the entire board. Amazingly, this is all we need to prove that any such board can be tiled!

At this point, let's draw an 8x8 board and apply the solution using a process called recursion, which is closely related to induction:



PowerPoint Slide to Chapter 5.

Practice problems (optional):

Carrano, Chapter 5:

1, 10, 11, 15

Also:

Prove that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ for n > 0.