

CSC 112 Homework 1 - Recursion Name _____

Due Date: Start of class, 10/7/2009

1. For the following problem descriptions, suggest the appropriate base case condition(s) and why those are the appropriate conditions:

- a) A recursive function that computes the logical OR over a size n array of Boolean values
 - if the end of the array has been reached ($\text{position} \geq \text{size}$)
 - if $\text{array}[\text{position}] == \text{true}$ (stop checking because only need to find one true)
- b) A recursive function that, given an array and a set of indices x and y such that x is less than y and both are less than the size of the array, reverses all array entries between x and y . [Problem 3 on page 606 of book]
 - if ($x > y$) (once the indices pass over each other)
- c) A recursive function that checks for whether or not a word is a palindrome by checking whether or not the first and last letters are the same, returning false if they aren't the same, and recursively testing the rest of the word (minus the front and back letter) if they are.
 - if ($\text{front} > \text{back}$) (looked at all of array once indices pass over each other)
 - if ($\text{word}[\text{front}] \neq \text{word}[\text{back}]$) (guaranteed to not be a palindrome at this point)

2. Below you will find an implementation of a recursive technique for generating the n th value of the Fibonacci sequence, which are the numbers such that $F_n = F_{n-1} + F_{n-2}$ (1, 1, 2, 3, 5, 8, 13, 21, 34...).

```
int fibonacci(int n)
{
    if ((n == 0) || (n == 1)) return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

While this is an very simple-looking function to read and write, I claim it is not a very efficient function in terms of amount of work that is performed. Indicate whether you believe with me or not and why.

You should agree with this claim. The biggest problem with this function is it repeats a lot of work. Let's look at $\text{Fibonacci}(5)$:

$$\begin{aligned} \text{Fibonacci}(5) &= \text{Fibonacci}(4) + \mathbf{\text{Fibonacci}(3)}; \\ &= (\mathbf{\text{Fibonacci}(3)} + \text{Fibonacci}(2)) + (\text{Fibonacci}(2) + \text{Fibonacci}(1)); \\ &= (\text{Fibonacci}(2) + \text{Fibonacci}(1)) + (\text{Fibonacci}(1) + \text{Fibonacci}(0)) + \\ &(\text{Fibonacci}(1) + \text{Fibonacci}(0)) + \text{Fibonacci}(1). \\ &= ((\text{Fibonacci}(1) + \text{Fibonacci}(0)) + \text{Fibonacci}(1)) + (\text{Fibonacci}(1) + \\ &\text{Fibonacci}(0)) + (\text{Fibonacci}(1) + \text{Fibonacci}(0)) + \text{Fibonacci}(1). \end{aligned}$$

Importantly, note that in solving Fibonacci(4) to come up with part of the answer for Fibonacci(5) you have to solve Fibonacci(3), and then you re-compute Fibonacci(3) for its own part of the Fibonacci(5) answer. This problem gets even worse the larger value we send to Fibonacci.

3. Demonstrate the correctness of the factorial method outlined in class using the “three-checks” verification method described in class.

Here’s our factorial function:

```
int factorial(int n)
{
    if ((n == 0) || (n == 1)) return 1;
    else return n * factorial(n-1);
}
```

Three checks:

- Verify recursion drives towards base case
- Verify base cases are correct
- Assuming $n-1$ case works correctly, verify processing for n case gives right answer.

The recursion, from any integer, uses -1, so it is ensured to count down in values. Because it is subtracting just 1, it will eventually reach 1 or 0, the base cases. We need to handle 0 because it is legally defined ($0! = 1$).

The base cases both match their mathematical definition ($0! = 1$) and ($1! = 1$).

Assume that factorial($n-1$) works correctly. Our function return $n * \text{factorial}(n-1)$. We know by definition that factorial($n-1$) is the product of all integer values from 1 to $n-1$. Thus, we are computing $n * \text{the product of all values from 1 to } n-1$, which is the product of all integer values from 1 to n , which matches the definition of factorial.

4. Write a recursive method that computes the logical OR over an array of Booleans

I used an approach based on the array processing pattern of – test the first element, if that fails, work on the remainder of the array as an array itself with the recursive function (similar to how the **max** function was described in the lab).

```
bool logicalOr(bool* array, int size)
{
    if (size == 0) return false; // looked at every element
    else if (*array == true) return true; // first element
    is true, so whole OR is true
    else return logicalOr(array+1, size-1); // if first
    element is false, look in rest of array
}
```