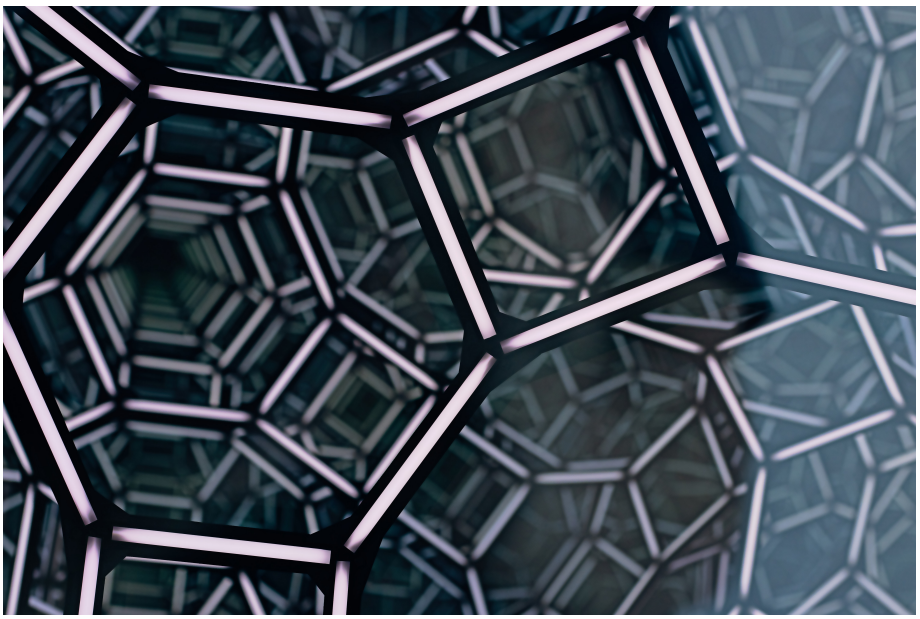
CS 202 - S22 - Assignment 7

Mr. Piotrowski

Recursion



1 Assignment Introduction

Welcome to Assignment 7! This assignment will be a little different than your previous assignments because there will be no use of classes. The only goal of this assignment is to practice recursion.

1.1 Brief Description of Recursion

Recursive Function (*programming*): Any function in which the function definition includes a call to itself.

Recursion is a method for solving problems. The closest related thing to recursion that you may find familiar is *iteration*. Recursion can be thought of similar to the for and while loops you use in your programs. In fact, anything that can be solved recursively can also be solved iteratively (*Church-Turing Thesis*).

Recursive functions solve problems by breaking larger problems into smaller problems. Before we describe more about how recursive functions work, let's look at an example.

Example: *The Fibonacci Sequence*

$$f(n) = f(n-1) + f(n-2)$$
where:
$$f(0) = 0, f(1) = 1$$

Observe the Fibonacci function above. This function only takes one number from the set of positive integers N (*where* $n \in N$) as a parameter. The C++ code for this function may look like:

```
1 unsigned int fibonacci(unsigned int n){
2     if(n <= 1){
3         return n;
4     }
5     return fibonacci(n - 1) + fibonacci(n - 2);
6 }
```

This function is recursive because the definition includes at least one call to itself (*on line 5*).

Any function can be recursive just by including a call to itself, but this does not make it useful. In computer science, a **finite** recursive function is really the only recursive function we care about. Any function that flat out calls itself without any extra work results in an infinite recursion:

```
1 void foo(){
2     foo();
3 }
```

Assuming this function is called, under what condition does the function stop calling itself? This recursive function is a infinite because there is no condition.

All finite recursive functions include the following 3 things:

1. At least one call to itself.
2. At least one base case.
3. At least one reduction operation.

A **base case** stops the recursion from continuing and the **reduction operation** is some work that takes the recursive function closer to the base case.

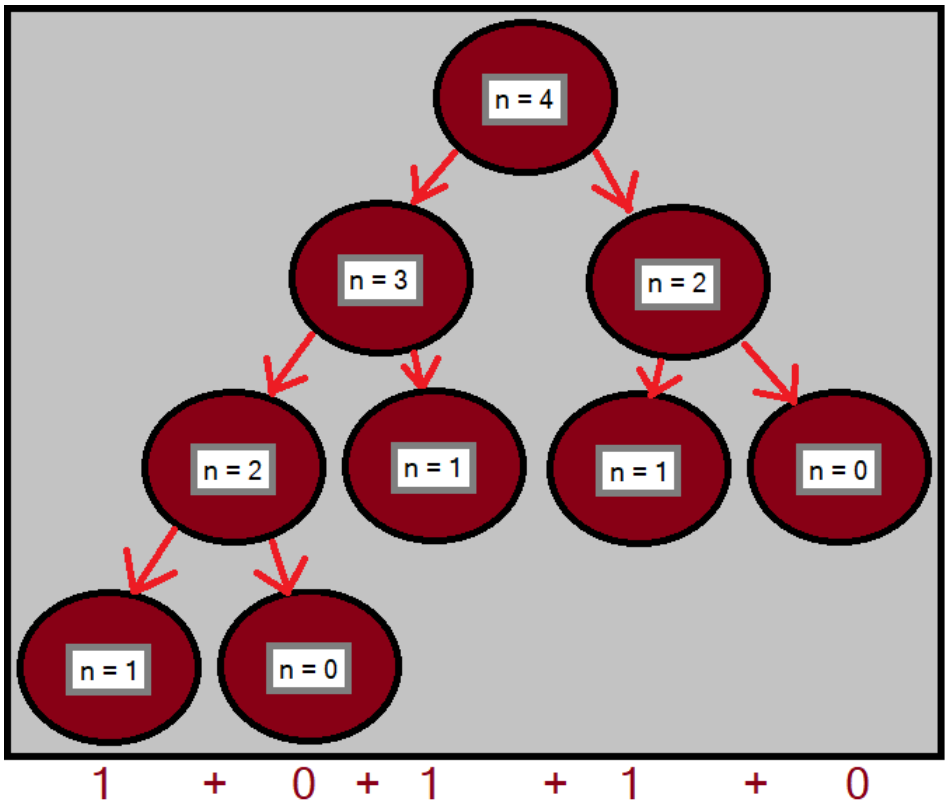
Looking back at the fibonacci function, this function has 2 base cases. In the case that n is either 1 or 0, the function returns 1 or 0 and does not recurse. The reduction operations are the $(n-1)$ and $(n-2)$ within the recursive function calls. These operations ensure that n will eventually reach the base cases of either 0 or 1.

As previously mentioned, recursion breaks larger problems into smaller ones. With the fibonacci function, what is the smallest problem? It is in fact the base cases, and the work done by this function is the addition of a bunch of ones and zeroes. Observe below:

$f(4) = f(3) + f(2)$
$f(4) = (f(2) + f(1)) + f(2)$
$f(4) = (f(2) + f(1)) + (f(1) + f(0))$
$f(4) = ((f(1) + f(0)) + f(1)) + (f(1) + f(0))$
$f(4) = ((1 + f(0)) + f(1)) + (f(1) + f(0))$
$f(4) = ((1 + 0) + f(1)) + (f(1) + f(0))$
$f(4) = ((1 + 0) + 1) + (f(1) + f(0))$
$f(4) = ((1 + 0) + 1) + (1 + f(0))$
$f(4) = ((1 + 0) + 1) + (1 + 0)$
$f(4) = 1 + 0 + 1 + 1 + 0$
$f(4) = 3$

Note this is a very inefficient way to solve this problem. Recursion always uses more resources than iteration, thus, recursion really should only be used if problems are difficult to solve with loops, or if the algorithm is faster than the loop solution (*for example, quick sort vs. bubble sort*).

Recursive functions can be represented with a *tree diagram*, which shows how the recursive function calls itself. Let us use the same example of $fibonacci(4)$:



We can see that this function is called **9 times** to completely solve the problem. We can also see that this tree is **binary** *or* **2-ary** as each recursive case will call itself *at most* twice (*each node has 2 children*). Functions that call themselves at most once can be represented with a **unary** or **1-ary** tree, and any recursive function that calls itself at most z times would make a **z-ary** tree.

2 Program Flow

Your goals are to write 5 recursive functions to solve the following problems:

1. Integer exponentiation. (n^x)
2. Counting the number of times a character c appears in a character array.
3. Popping Balloons and adjacent balloons of the same color in a 2D grid.

4. Counting the number of non-space characters in a sentence that is stored in a 2-dimensional character pointer.

5. Merging 2 alphabetical character arrays in alphabetical order (*whilst also counting the total number of chars*).

There is no particular flow. Each function you write will be tested individually on multiple test cases.

3 Coding The Assignment

Important Note: The codegrade for this assignment automatically checks to see if the function you wrote is recursive. If you code a solution to the function that is not recursive, codegrade will not give you points! In order to check your functions for recursive function calls, I had to write a python script that reads your code. The existence of block comments (*comments written like the one below:*)

```
1 /* This is a block comment */
```

make it very difficult for my script to verify that your function is truly recursive, so **you may NOT use block comments for this assignment.** If there are any block comment symbols (/* *and* */), codegrade will not give you any points. Please replace all block comments with line comments (//).

There are 3 files:

- recursiveFuncs.h
- recursiveFuncs.cpp (**your code here**)
- main.cpp

The function skeletons are already provided for you in the recursiveFuncs.cpp file. You may not alter the parameters for any of the functions.

For each function, I have included the number of lines I used in my solution. This includes lines that are just brackets, so, the number of lines that actually do something in my functions are anywhere between 1/3 and 1/2 the number you will see. The number is there to remind you **not to overthink**. If your functions start going way over the number of lines I used, you are likely overthinking and need to re-approach the problem.

3.1 Integer Exponentiation

INSTRUCTOR SOLUTION LINE COUNT: **4**

```
unsigned int integerPow(unsigned int num, unsigned int pow);
```

Write a function to calculate num^{pow}. The first parameter is the number and the second parameter is the exponent. The function will be called like this:

```
unsigned int result = integerPow(2,3); // 2^3 = 8
```

Exponentiation is just the repeated multiplication of a number. 2^3 is just $2*2*2$. That is also just $(2*2^0)*(2*2^0)*(2*2^0)$ which gives you $(2*1)*(2*1)*(2*1)$.

1. What are your base case(s)?
2. What are your reduction operation(s)?
3. Draw the tree diagram for this function when called for 3^4.
4. What kind of tree does your function make?

3.2 Counting Specific Characters

INSTRUCTOR SOLUTION LINE COUNT: **11**

```
unsigned int countCharacter(char *str, char c, unsigned int
    currentIndex, unsigned int strLen, bool left, bool right);
```

Write a function to count the number of times a specific character c appears in a character array. You will be provided the character array, the character to search for, the current index that is being looked at, the length of the character array, and the 2 directions to search as parameters. This function should work no matter what index is provided as a starting position. For example:

```
const unsigned int strLen = 30;
char str[strLen] = "hello and welcome to my guide!";

cout << countCharacter(str, 'm', 0, strLen, true, true) << endl;
cout << countCharacter(str, 'm', 5, strLen, true, true) << endl;
cout << countCharacter(str, 'm', 20, strLen, true, true) << endl;
cout << countCharacter(str, 'm', 29, strLen, true, true) << endl;

cout << countCharacter(str, 'e', 0, strLen, true, true) << endl;
cout << countCharacter(str, 'e', 5, strLen, true, true) << endl;
cout << countCharacter(str, 'e', 20, strLen, true, true) << endl;
cout << countCharacter(str, 'e', 29, strLen, true, true) << endl;
```

The code above should print:

```
2
2
2
2
4
4
4
4
```

The purpose of the bools is to help you stop the recursion. If the left bool is true, then you must still count all the characters to the left of the currentIndex. If the right bool is true, then you must still count all the characters to the right of the currentIndex.

1. What are your base case(s)?
2. What are your reduction operation(s)?
3. Draw the tree diagram for this function when called as countCharacter("las vegas", 's', 0, strLen).
4. What kind of tree does your function make?

3.3 Popping Balloons

INSTRUCTOR SOLUTION LINE COUNT: **10**

```
void popBalloons(char **grid, unsigned int currentHeight, unsigned
    int currentWidth, unsigned int height, unsigned int width, char
     balloon);
```

Given a 2d grid of balloons, the coordinates of a specific balloon in the grid, and the color of a balloon, pop that balloon and all balloons of the same color that are adjacent to the balloon (*up, left, right, down*). Observe the grid below with yellow balloons, blue balloons, and red balloons:

```
  |  0  1  2  3  4  5
----------------------
0 | [Y][Y][Y][Y][Y][Y]
1 | [Y][B][B][B][B][Y]
2 | [Y][B][R][R][B][Y]
3 | [Y][B][B][B][B][Y]
4 | [Y][Y][Y][Y][Y][Y]
----------------------
  |
```

If this function were to be called like this:

```
popBalloons(grid, 3, 3, 5, 6, 'B');
```

The blue balloon at [3][3] would pop and set off a chain reaction to pop every blue balloon touching the initial balloon. Those other blue balloons would then pop their adjacent neighbors too, etc.

```
  |  0  1  2  3  4  5
----------------------
0 | [Y][Y][Y][Y][Y][Y]
1 | [Y][ ][ ][ ][ ][Y]
2 | [Y][ ][R][R][ ][Y]
3 | [Y][ ][ ][ ][ ][Y]
4 | [Y][Y][Y][Y][Y][Y]
----------------------
  |
```

The function does not consider diagonals. Example below:

```
  |  0  1  2  3  4
-------------------
0 | [Y][Y][Y][Y][Y]
1 | [Y][Y][B][B][B]
2 | [Y][B][Y][Y][Y]
-------------------
  |
```

```
popBalloons(grid, 2, 4, 3, 5, 'Y');
```

```
  |  0  1  2  3  4
-------------------
0 | [Y][Y][Y][Y][Y]
1 | [Y][Y][B][B][B]
2 | [Y][B][ ][ ][ ]
-------------------
  |
```

The equivalent to popping a balloon is simply overwriting the character in the grid with a space ' '. This will help you stop the recursion.

1. What are your base case(s)?
2. What are your reduction operation(s)?
3. Draw the tree diagram for this function when called for:

   ```
     |  0  1  2  3  4
   -------------------
   0 | [Y][Y][Y][Y][Y]
   1 | [Y][Y][B][B][B]
   2 | [Y][B][Y][Y][Y]
   -------------------
     |
   ```

   ```
   popBalloons(grid, 2, 4, 3, 5, 'Y');
   ```

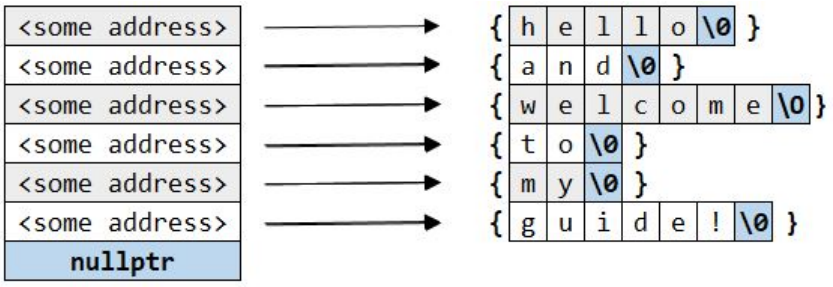
4. What kind of tree does your function make?

3.4 Counting the non-space Characters in a Sentence

INSTRUCTOR SOLUTION LINE COUNT: **7**

```
unsigned int countLettersInSentence(char **sentence, unsigned int
    currentWord, unsigned int currentChar);
```

This seems to be like the other function you already coded but it is not quite the same. This is because of how the sentence is stored. The sentence for this function is stored as a 2D character array. Words are terminated by NULL characters and the sentence is terminated with a nullptr. For example, the sentence "hello and welcome to my guide!" Would be stored in a character double pointer that has 6 words + 1 extra space for a nullptr. Each pointer stored in the double pointer will point to a different character array with a different length. See below:



You can guarantee that the currentWord and currentIndex will always be passed as 0 and 0 on the first function call. Function is called like this:

```
cout << countLettersInSentence(sentence, 0, 0);
```

1. What are your base case(s)?
2. What are your reduction operation(s)?
3. Draw the tree diagram for this function when called for the sentence "My brand!"
4. What kind of tree does your function make?

3.5 Alphabetical Merge

INSTRUCTOR SOLUTION LINE COUNT: **28**

```
unsigned int alphaMerge(char *&dest, unsigned int currentSize, char
     *arr1, char *arr2, unsigned int i1, unsigned int i2);
```

Given two null-terminated character arrays that are already in alphabetical order, make a new character array which is the two smaller arrays combined in alphabetical order. For example:

```
char *newArray = nullptr;
unsigned int size = alphaMerge(newArray, 0, "acegi", "bdfhj", 0, 0)
    ;
cout << size << endl;
for(unsigned int i = 0; i < size; i++){
    cout << newArray[i];
}
cout << endl;
```

The code above would print:

```
10
abcdefghij
```

10, the integer returned by the function, would be the size of the new array. Also, the newArray pointer would contain the resulting char array. The first parameter is the pointer to store the new char array in, the currentSize is the current size of the destination array, the arr1 and arr2 parameters are the two character arrays to merge, and the two i1 and i2 parameters are the current index of each character array, respectively. This function will always be called to start with an empty destination character array and 0 for all unsigned int parameters.

1. What are your base case(s)?
2. What are your reduction operation(s)?
3. Draw the tree diagram for this function when called for the two character arrays "aac" and "bd"
4. What kind of tree does your function make?

4 Compiling and Submitting

Your program should compile with the following:

```
g++ -std=c++11 main.cpp recursiveFuncs.cpp
```

Run your program with:

```
./a.out
```

I strongly recommend commenting out sections of the main.cpp to individually test your functions. You can use block comments in main.cpp, just not in recursiveFuncs.cpp.

Submit **only** your recursiveFuncs.cpp program to CodeGrade.

5 Comprehension Questions

For all 5 recursive functions you wrote, you should have answered 4 questions for each. Please document these in a PDF form and turn them in. If your class is remote, please upload the PDF to webcampus. If your class is in-person, check with your instructor to see if you can turn in the documents in-person or on webcampus.