INTRODUCTION TO COMPUTER PROGRAMMING
BERKELEY CITY COLLEGE
FALL 2019 -- LAB 2

It is very important that you attend lectures and labs. Many of these problems will be discussed in class.

IMPORTANT NOTE #0: Many of these questions involve figuring out working algorithms for solving complex problems. I strongly, strongly, STRONGLY recommend designing these algorithms on paper and verifying that they're correct BEFORE you try to write them in code.

IMPORTANT NOTE #1: You ALREADY know most of the C++ syntax required to solve these problems. Several of these problems are relatively difficult; the difficulty is less about knowing the syntax and more about correctly designing the algorithms. (sort of like how the Karel questions from lab 0 were difficult).

IMPORTANT NOTE #2: GET STARTED EARLY. GET. STARTED. EARLY. *GET STARTED EARLY*. YOU WILL NOT BE ABLE TO FINISH THIS ALL ON THE LAST DAY OR THE LAST WEEK.

IMPORTANT NOTE #3: I will be opening up separate turn-in spots on Canvas for each question. After the second week I will, upon request, provide feedback on any two of these questions (with each *part* of the arrays question counting as a question for these purposes). Once you have received feedback you *cannot* change your response, so only ask if you are *certain* you are done with the problem.

Use good programming practices. Display messages so that the results are easily understandable. If the question asks for a particular format of output, please follow it to the letter.

1. **Stars**
   Write a program that prints a diagram of the function 2^n, as follows.

```
*
**
****
********
****************
********************************
****************************************************************
********************************
****************
********
****
**
*
```

   Your program should contain only one '*' character.

2. **Vowels and consonants**
   Write a program that reads a word and prints the number of vowels and consonants in the word. For this exercise assume that 'a', 'e', 'i', 'o', 'u', and 'y' are vowels. For example, if the user enters the input "Harry", the program should print "The word Harry contains 2 vowels and 3 consonants."

3. **Fibonacci sequence**
   Write a program to compute the n^{th} Fibonacci number. Fibonacci numbers are recursively defined as follows:

$$f_o = 0$$
$$f_1 = 1$$
$$f_k = f_{k-1} + f_{k-2}$$

   where f_i is the i^{th} Fibonacci number.
   (The zeroth and first Fibonacci numbers are defined as 0 and 1, respectively. Knowing the first two Fibonacci numbers we can calculate the second using the formula $f_2 = f_1 + f_0 = 1$, and so on.)

   note: You must find a *non-recursive* solution. If you find a recursive solution, please *also* include a non-recursive one.

   (I'll give a few extra credit points if you implement both a recursive and a non-recursive solution, but not a huge amount).

4. **Karel version 0.2**

Create a 15x15 two dimensional grid (using whatever data structure you prefer – arrays, vectors, strings, whatever). Each square on the grid should be marked with a – character. Place a robot, marked with a O, on the bottom left square of this grid, facing right. Print the board, then provide a prompt for the user and ask for input. Your program must take the following input:

```
move();
turnLeft();
putBeeper();
pickBeeper();
quit();
```

Each time you take valid input from the user, you should print the board.

move(); should move the robot one square in the direction they are facing. turnLeft(); should adjust the robot's facing accordingly. putBeeper() should put a beeper on the grid, marked with the letter X. If Karel is standing on a beeper, use the character @ to represent the combination. If the user places a beeper on a square that already has a beeper, the display should be unchanged (you don't have to worry about multiple beepers on one square for this version). If the user enters pickBeeper(); when the robot is on a beeper, remove the beeper. If the user enters pickBeeper(); when there is no beeper, the program should quit after displaying a message saying the robot has crashed. Likewise, if the robot move();s into a wall, the program should quit after displaying a crash message.

If the user enters quit();, your program should quit.

If the user enters any other string, print "syntax error" and re-display the prompt.

Note: You **MUST** have your user format input **EXACTLY** as given in this question. move(); is a valid move, but move or move() is a syntax error.

5. Histograms

a) Write a program to generate the histogram for the random number generator. Create 10 counters with the following limits:

Counter	Limit
1	0 <= x < 0.1
2	0.1 <= x < 0.2
3	0.2 <= x < 0.3
4	0.3 <= x < 0.4
5	0.4 <= x < 0.5
6	0.5 <= x < 0.6
7	0.6 <= x < 0.7
8	0.7 <= x < 0.8
9	0.8 <= x < 0.9
10	0.9 <= x < 1.0

First initialize each counter to zero. Generate a random number between 0 and 1. Increment the corresponding counter by 1. Repeat this 10,000 times. Whenever the random number generated falls in a particular range increment the corresponding counter bin by one. Plot the counter frequencies (i.e. the histogram)

Use hash marks (#) to represent the histogram bars. Separate each bar in the histogram with | symbols. Mark the top of each bar with - symbols. Place one space betwen each bar. Scale your histogram such that the largest value is no more than 50 hash marks tall.

b) Modify the program to plot the histogram of the *sum* of 10 random numbers. Create 10 counters with the following limits:

Counter	Limit
1	0 <= x < 1
2	1 <= x < 2
3	2 <= x < 3
4	3 <= x < 4
5	4 <= x < 5
6	5 <= x < 6
7	6 <= x < 7
8	7 <= x < 8
9	8 <= x < 9
10	9 <= x < 10

6. **Square Root**

Write a function to determine the square root of a number. The square root of a number can be approximated by repeated calculation using the formula

$$NG = 0.5(LG + N/LG)$$

where NG stands for the next guess and LG stands for the last guess. The loop should repeat until the difference between NG and LG is less than 0.00001. Use an initial guess of 1.0. Write a driver program to test your square root function.

(This is called *Newton's Method* and can approximate square roots relatively rapidly, even if you start with a terrible guess).

7. **Arrays**

Write the following array functions. Use integer arrays. Once you have written your functions, write a driver program to test them. Consider edge cases -- what if your functions are used on arrays with only one value? What if your functions are used on empty arrays? What if your functions are used on arrays wherein all items are the same value?

You will be graded both on the correct functioning of your functions, and also on how extensively your driver program tests them.

a. Suppose, `arr[] = {1, 4, 6, 5, 2, 7, 10, 4}`

Search for a given value in the first *n* items of an array and return the **index** of the location. Return -1 if not found in the array. You can safely assume that *n* is less than or equal to the size of the array.

```
int search(int arr[], int n, int val, bool left);
```

Using the above array `arr`, `search(arr, 7, 6, true)` should return 2, `search(arr, 8, 8, true)` should return -1, and `search(arr, 2, 5, true)` should return -1.

If there are multiple copies of one value in an array, use the value of the `left` variable to determine which is returned.

Using the above array `arr`, `search(arr, 8, 4, true)` should return 1, while `search(arr, 8, 4, false)` should return 7.

If you search from the right and can't find the value in the range (maxIndex, maxIndex - n), return -1 even if the item exists in the range (0, maxIndex - n - 1) [both ranges are inclusive]

b. Reverse the contents of the array.

```
void reverse(int arr[], int arraySize);
```

Using the above array `arr`, `reverse(arr, 7)` should return result in arr contents rearranged as follows: `arr = {10, 7, 2, 5, 6, 4, 4, 1}`

c. Rearrange the array such that all the odd elements in the array are in the beginning of the array.

```
void oddFirst(int arr[], int arraySize);
```

Using the above array `arr`, `oddFirst(arr, 7)` should return result in arr contents rearranged as follows: `arr = {1, 5, 7, 4, 4, 6, 2, 10}`

d. Write a function that finds all *local maxima* in an array. A local maximum is an array item that is higher in value than both of the array values on each side. If an array item is at the start or end of the list, it is a local maximum if it is higher than its neighbor. An item in a single-item array is always a local maximum.

Your function should take two arrays as input: one containing the source array, and the other containing an output array, initially empty, in which you place the **indices** of each of the local maxima from the source array. It should also take an int indicating the size of the array.

A *plateau* is a set of array items with the same value, where the neighbors on each side of the plateau are lower in value than the array items making up the plateau. If you find plateaus in the source array, place the index of the *rightmost* item in the plateau in your output array.

Your function should *return* the value of the *global maximum*, i.e. the largest item in the entire list.

e. Write a function that returns the length of the *longest run of the same value* in an array. If there are no array items that have the same value as their neighbors, return 0.