EC327 – "Introduction to Software Engineering"
<u>Lab5 – Arrays</u>
Mo, 10/05/15 – Fr, 10/09/15

In Lab5, **we define three problems** (P1, P2, P3) of similar complexity. To pass Lab5, **you choose one problem** and try to solve it. You are not required to solve all four problems!

The problems' solutions focus on the practical utilization of C++ **data types**, **user inputs, conditional branches (**`IF-ELSE`**), loops (**`DO-WHILE`**,** `FOR`**,** `WHILE`**), (recursive) functions, header files**, and **arrays.**

In your designated lab hours you should demonstrate and explain your solution approach of the chosen problem. You are not required to work outside lab hours! We recommend starting early. For example, you can solve the problem at home, come to lab, demonstrate your solution, and get checked off. Ensure that your code compiles and executes properly in the lab.

For a given problem, you will download the provided test framework and write the corresponding .h and .cpp files. Make sure that your code compiles with the provided framework and runs as expected before getting checked off.

Read the problem statements carefully. Decide which problem to solve, and decide when to start. Do not hesitate to ask any clarifying questions.

ENJOY!

P1 – Median and Array reversal

p1-test.cpp, compute.cpp, compute.h

In this problem you have to implement functions that calculate the median of an array of integers and perform specific array manipulations (see below). You may assume that the input array is **sorted in ascending order**.

Structure your code into the following three files:

- `compute.cpp (partly provided)`

  There are three functions:

  - `double findMedian(int input [], int size)`

    This function should return the median of an input array given an array and its size

  - `void magic(int input[], int size, double result_magic[], int & result_size)`

    This function should reverse the input array and subtract each element by the median and store the result inside `result_magic`, `result_size` would store the number of valid elements in the result, which should be the same as `size` in this case.

  - `void printStats(int input[], int size, double median, int result_magic[], int result_count)`

    This function takes in the inputs and results and prints it out to console (It's provided, and you should not have to do anything with it)

- `compute.h`

  Prototype the functions here.

- `p1-test.cpp (provided for testing)`

  It defines the `main()` function for testing

Your output with the provided test file should look like this:

```
Input: 5, 6, 6, 6, 6, 6, 6, 6, 7
Median: 6
Magic result: 1, 0, 0, 0, 0, 0, 0, 0, -1

Input: 0, 10, 11, 12, 12, 13, 13, 14, 14, 15
Median: 12.5
Magic result: 2.5, 1.5, 1.5, 0.5, 0.5, -0.5, -0.5, -1.5, -2.5, -12.5

Input: 0, 1, 50, 51, 52, 53, 54, 55, 57, 100, 110
Median: 53
Magic result: 57, 47, 4, 2, 1, 0, -1, -2, -3, -52, -53

Input: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Median: 6.5
Magic result: 5.5, 4.5, 3.5, 2.5, 1.5, 0.5, -0.5, -1.5, -2.5, -3.5, -4.5, -5.5
```

P2 – Mode Computation

p2-test.cpp, mode.cpp, mode.h

In this problem you have to implement a function that finds the mode(s) of an **unsorted** array of integers.
Structure your code into the following three files:

- `mode.cpp (partly provided)`
  There are three functions:
  - `findModes(int input[], int size, int & frequency, int array_result[], int & result_count){`
    This function takes in the inputs, finds the mode(s) - there can be more than one mode - and stores the results inside `array_result`, with `result_count` denoting the number of modes, and `frequency` the frequency of the modes.
  - `void printModes(int input[], int size, int frequency, int result[], int result_count)`
    This function takes in the inputs and results and prints it out to console (It's provided, and you should not have to do anything with it)
- `mode.h`
  Prototype the functions here.
- `p2-test.cpp (provided for testing)`
  It defines the `main()` function for testing

Your output with the provided test file should look like this:

```
Input: 2, 3, 4, 4, 3, 1, 2, 1, 2, 3
Modes: 2, 3
Frequency : 3

Input: 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
Modes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Frequency : 1

Input: 1, 3, 4, 9, 7, 8, 12, 13, 2, 3
Modes: 3
Frequency : 2

Input: 2, 3, 4, 4, 5, 1, 2, 1, 5, 3
Modes: 1, 2, 3, 4, 5
Frequency : 2
```

P3 – Palindrome

`palindrome_functions.cpp`, `palindrome_functions.h`, `palindrome.cpp`

The user inputs a string of characters and the program checks if the entered string is a palindrome (`http://en.wikipedia.org/wiki/Palindrome`).

Structure your code into the following three files:

- `palindrome_functions.cpp`
  Define two functions:
  - `bool is_palindrome(char*)` which returns true if the character string is a palindrome. Otherwise the function returns false.
  - `bool is_case_palindrome(char*)` which returns true if the character string is a palindrome, where one string is the other string with letter case inverted (For example: aaaAAA, Fe3Jj3Ef). Otherwise the function returns false.

- `palindrome_functions.h`
  Prototype the functions here.

- `palindrome.cpp`
  Here define the `main()` function. Also, use proper `#include` statements.

The program should only terminate if the string contains a numeric value (`do-while` loop!).

<u>Sample Output:</u>

```
String: aaabbbaaa
The string aaabbbaaa is a palindrome.

String: atcg
The string atcg is not a palindrome.

String: race car
The string race car is a palindrome.

String: eC327 723cE
The string is a palindrome with inverted case.
Bye Bye!
```