NYU, Tandon School of Engineering
Bridge to CS — Fall 2017

Homework #7
Due by Friday 10/27 11:55pm

Submission instructions:

1. Pay special attention to the style of your code. Indent your code correctly, choose meaningful names for your variables, define constants where needed, choose most suitable control statements, break down your solutions by defining functions, etc.
2. You should submit your homework in the NYU Classes system.
3. For this assignment you should turn in 8 files, one for each question:
   - Seven '.cpp' files (for questions 1, 2, 3, 4, 6, 7 and 8). Name these files: 'YourNetID_hw7_q1.cpp', 'YourNetID_hw7_q2.cpp', etc.
     For each of these questions, create a main function that calls the other functions you were asked to implement in the question.
   - One '.pdf' file (for question 5). Name this file 'YourNetID_hw7_q5.pdf'.

Question 1:

Give a **recursive** implement to the following functions:

a. `void printTriangle(int n)`
   This function is given a positive integer `n`, and prints a textual image of a right triangle (aligned to the left) made of n lines with asterisks.
   For example, `printTriangle(4)`, should print:

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

b. `void printOpositeTriangles(int n)`
   This function is given a positive integer `n`, and prints a textual image of a two opposite right triangles (aligned to the left) with asterisks, each containing n lines.
   For example, `printOpositeTriangles(4)`, should print:

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

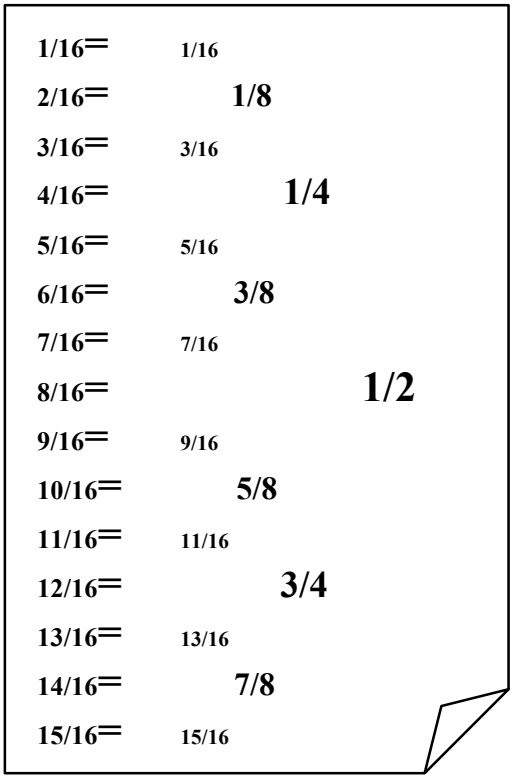
c. `void printRuler(int n)`

This function is given a positive integer `n`, and prints a vertical ruler of $2^n - 1$ lines. Each line contains '-' marks as follows:

- The line in the middle ($\frac{1}{2}$) of the ruler contains *n* '-' marks
- The lines at the middle of each half ($\frac{1}{4}$ and $\frac{3}{4}$) of the ruler contains (*n*-1) '-' marks
- The lines at the $\frac{1}{8}$, $\frac{3}{8}$, $\frac{5}{8}$ and $\frac{7}{8}$ of the ruler contains (*n*-2) '-' marks
- And so on …
- The lines at the $\frac{1}{2^k}$, $\frac{3}{2^k}$, $\frac{5}{2^k}$, …, $\frac{2^k-1}{2^k}$ of the ruler contains *1* '-' mark

For example, `printRuler(4)`, should print (only the blue marks):

```
-
- -
-
- - -
-
- -
-
- - - -
-
- -
-
- - -
-
- -
-
```



Hints:

1. Identify what `printRuler(n-1)` is **supposed** to print, and use that in order to define how to print the ruler of size `n`.
2. You may want to have more than one recursive call

Question 2:

Give a **recursive** implement to the following functions:

a. `int sumOfSquares(int arr[], int arrSize)`
   This function is given `arr`, an array of integers, and its logical size, `arrSize`. When called, it returns the sum of the squares of each of the values in `arr`.
   For example, if `arr` is an array containing [2, -1, 3, 10], calling `sumOfSquares(arr, 4)` will return 114 (since, $2^2 + (-1)^2 + 3^2 + 10^2 = 114$).

b. `bool isSorted(int arr[], int arrSize)`
   This function is given `arr`, an array of integers, and its logical size, `arrSize`. When called, it should return `true` if the elements in `arr` are sorted in an ascending order, or `false` if they are not.

Question 3:

Write two **recursive** versions of the function `minInArray`. The function will be given a sequence of elements and should return the minimum value in that sequence. The two versions differ from one another in the technique we use to pass the sequence to the function.

In version 1 – The prototype of the function should be:
`int minInArray1(int arr[], int arrSize)`
Here, the function is given `arr`, an array of integers, and its logical size, `arrSize`.
The function should find the minimum value out of all the elements in positions:
0, 1, 2, …, arrSize-1.

In version 2 – The prototype of the function should be:
`int minInArray2(int arr[], int low, int high)`
Here, the function is given `arr`, an array of integers, and two additional indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.
The function should find the minimum value out of all the elements in positions:
low, low+1, …, high.

Use the following main to check your program:

```
int main() {
    int arr[10] = {9, -2, 14, 12, 3, 6, 2, 1, -9, 15};
    int res1, res2, res3, res4;

    res1 = minInArray1(arr, 10);
    res2 = minInArray2(arr, 0, 9);
    cout<<res1<<" "<<res2<<endl;  //should both be -9

    res3 = minInArray2(arr, 2, 5);
    res4 = minInArray1(arr+2, 4); //arr+2 is equivalent to &(arr[2])
    cout<<res3<<" "<<res4<<endl;  //should both be 3

    return 0;
}
```

Question 4 (taken from Ch14 problem 6 in text):

The game of "Jump It" consists of a board with *n* positive integers in a row, except for the first column, which always contains zero. These numbers represent the cost to enter each column. Here is a sample game board where *n* is 6:

0	3	80	6	57	10

The object of the game is to move from the first column to the last column with the lowest total cost.

The number in each column represents the cost to enter that column. You always start the game in the first column and have two types of moves. You can either move to the adjacent column or jump over the adjacent column to land two columns over. The cost of a game is the sum of the costs of the visited columns.

In the board shown above, there are several ways to get to the end. Starting in the first column, our cost so far is 0. We could jump to 80, then jump to 57, and then move to 10 for a total cost of 80 + 57 + 10 = 147. However, a cheaper path would be to move to 3, jump to 6, then jump to 10, for a total cost of 3 + 6 + 10 = 19.

Write a **recursive** function that solves this problem and returns the lowest cost of a game board represented and passed as an array.

Note: your function shouldn't output the actual sequence of jumps, only the lowest cost of this sequence.

Question 5:

a. Use mathematical induction to prove that for any positive integer n, 3 divide $n^3 + 2n$ (leaving no remainder).
   Hint: you may want to use the formula: $(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$.

b. Use strong induction to prove that any positive integer n ($n \geq 2$) can be written as a product of primes.

Question 6:

Implement an iterative (non-recursive) implementation for the binary search algorithm. The prototype of the function should be:

```
int binarySearch(int arr[], int arrSize, int val)
```

The function is given `arr`, an array of integers that are ordered in an ascending order, and its logical size, `arrSize`, and an integer value `val` to search for. When called, it returns the index of `val` in `arr`.

Notes:

- If `val` does not exist in `arr`, `binarySearch` should return -1.
- If `val` appears more than once in `arr`, `binarySearch` can return any one of the indices where `val` appears in.

Question 7:

Implement the function:

`int findChange(int arr01[], int arr01Size)`

This function is given `arr01`, an array of integers containing a sequence of 0s followed by a sequence of 1s. The function also gets its logical size, `arr01Size`. When called, it returns the index of the first 1 in `arr01`.

For example, if `arr01` is an array containing [0, 0, 0, 0, 0, 1, 1, 1], calling `findChange(arr01, 8)` will return 5.

Note: Pay attention to the running time of your function. If `arr01` is of size n, an efficient implementation would run in logarithmic time (that is $\Theta(log_2(n))$).

Question 8 (based on Ch7 problem 6 in text):

Write the function:

`void insertionSort(int a[], int aSize)`

That given `a`, an array of integers, and its logic size, `aSize`. The function implements the insertion-sort algorithm in order to sort the elements of `a` in an ascending order.

The insertion-sort algorithm works as follows:

It "picks up" successive elements from the array, and inserts each of these into the correct position in an already sorted subarray (at one end of the array we are sorting).

The array to be sorted is divided into a sorted subarray and an unexamined subarray. Initially, the sorted subarray is empty. Each element of the unexamined subarray is picked and inserted into its correct position in the sorted subarray.

The implementation involves an outside loop that selects successive elements in the unsorted subarray, and a nested loop that inserts each element in its proper position in the sorted subarray.

The outside loop will maintain the following invariant: At the end of the i-th iteration, the elements, originally at a[0], a[1], …, a[i-1], are sorted (among each other)

For example:

Initially, the sorted subarray is empty, and the unsorted subarray is the entire array:

unsorted									
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

1st iteration:

Pick the first element from the unsorted subarray, a[0] (that is 8), and place it in the first position. The inside loop has nothing to do in this first case.

At the end of the iteration, the array and subarrays look like this:

sorted	unsorted								
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

2nd iteration:

Pick the first element from the unsorted subarray, a[1] (which has value 6). Insert this into the sorted subarray in its proper position. These are out of order, so the inside loop must swap values in position 0 and position 1.

At the end of the iteration, the array and subarrays look like this:

sorted		unsorted							
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	12	10

Note that the sorted subarray has grown by one entry.

3rd iteration:

Repeat that process again. That is, for the first unsorted subarray entry, a[2], finding a place where a[2] can be placed so that the subarray remains sorted. Since a[2] is already in place, that is, it is larger than the largest element in the sorted subarray, the inside loop has nothing to do. The result is as follows:

Sorted			unsorted						
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	12	10

4th iteration:

Again, pick the first unsorted array element, a[3]. This time the inside loop has to swap values until the value of a[3] is in its proper position. This involves some swapping:

sorted				unsorted					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10 <- - > 2		16	4	18	14	12	10

sorted				unsorted					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8 < - - > 2		10	16	4	18	14	12	10

sorted				unsorted					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6 < - - > 2		8	10	16	4	18	14	12	10

At the end of the iteration, the result of placing the 2 in the sorted subarray is:

sorted				unsorted					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
2	6	8	10	16	4	18	14	12	10

The algorithm continues in this fashion until the unsorted subarray is empty and the sorted subarray has all the original array's elements.