Friday, May 27, 2022.

Data Structures and Algorithms LAB – Spring 2022
(BS-IT-F20 Morning & Afternoon)

Lab # 7

Instructions:

- Attempt the following tasks exactly **in the given order**.
- You must complete all tasks individually. Absolutely NO collaboration is allowed. Any traces of plagiarism/cheating would result in an **"F"** grade in this course and lab.
- **Indent** your code properly.
- Use meaningful variable and function names. Use the **camelCase** notation.
- Use meaningful prompt lines/labels for all input/output.
- Do NOT use any **global** or **static** variable(s). However, global named constants may be used.
- Make sure that there are **NO** ***dangling pointers*** or ***memory leaks*** in your programs.

Task # 0 *(Pre-Requisite)*

Implement a **recursive** C++ function which takes an array of integers (`arr`) and the starting (`start`) and ending (`end`) indices of a *portion* (part) of this array, and returns the **index of the largest element** present in that portion of the array `arr`. The prototype of your function should be:

```
int findLargestIndex (int* arr, int start, int end)
```

For example, the function call `findLargestIndex(arr,3,8)` should determine and return the ***index*** of the largest element present in the array `arr` between the indices `3` and `8` (both inclusive).

Task # 1

Implement a **recursive** C++ function which takes an array of integers (`arr`) and the starting (`start`) and ending (`end`) indices of a *portion* (part) of this array, and **reverses the contents of that portion of the array**. The prototype of your function should be:

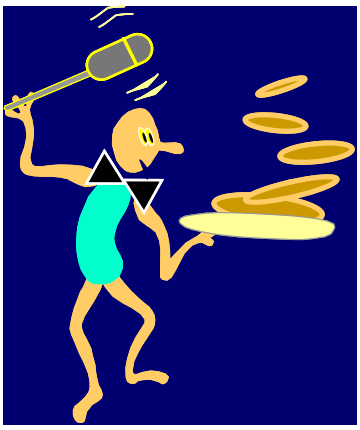
```
void reverseSubArray (int* arr, int start, int end)
```

Task # 2

In this task you are going to implement a new sorting algorithm called **Pancake Sort** (*yummy!!*). Before looking at the description of this algorithm, let's talk about its background:

The "Pancake Problem" was originally posed in an article of *American Mathematical Monthly* ("Harry Dweighter" a.k.a. Jacob Goodman, Amer. Math. Monthly, Vol. 82, No. 1, 1975, Page 1010), as follows:

"The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them so that the smallest winds up on top, and so on, down to the largest at the bottom. This is done by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are *n* pancakes, what is the maximum number of flips (as a function *f*(*n*) of *n*) that I will ever have to use to rearrange them?**"**



In this lab we will not concern ourselves with the computation of *f*(*n*) as described above, rather we will try to implement the sorting algorithm described in the previous paragraph. Each pancake can be represented by an integer, indicating its diameter. We can also visualize a stack of pancakes in the form of an array. The top-most pancake in the stack will be the left-most value in our array, and the bottom-most pancake will be the right-most value in our array. **So, the problem of sorting a stack of pancakes can be thought of as sorting an array of positive integers into increasing order.**

At any point during the sorting, you can only flip (reverse) some number of pancakes from the top of the stack. This means that at any point during the sorting, you can modify the contents of the array ONLY by **reversing** some prefix (left-most elements) of the array (**see the function which you implemented above in Task # 1**). You are NOT allowed to modify the contents of the array in any other way.

For example, if the array to be sorted is **[4 8 7 2]**, the algorithm will proceed as follows:

1. Find the index of the largest element from the whole array (index = 1) (**Task # 0**).
2. Reverse the subarray from index 0 to index of the largest element found in previous step (i.e. from index 0 to 1). Array becomes: **[8 4 7 2]**. (*Note that the largest element has been moved at the start*) (**Task # 1**).
3. Reverse the subarray containing the first 4 elements (i.e. the whole list): **[2 7 4 | 8]**. (*Note that the largest element has been moved to its correct position*) (**Task # 1**).

4. Find the index of the largest element among the remaining elements i.e. the first 3 elements (index = 1).
5. Reverse the subarray from index 0 to index of the largest element found in previous step (i.e. from index 0 to 1). Array becomes: `[7 2 4 8]`.
6. Reverse the subarray containing the first 3 elements of the array. Array becomes: `[4 2 | 7 8]`. (*Note that the second largest element has also been moved to its correct position*).
7. Find the index of the largest element among the remaining elements i.e. the first 2 elements (index = 0).
8. Reverse the subarray from index 0 to index of the largest element found in previous step (i.e. from index 0 to 0). Array remains: `[4 2 7 8]`.
9. Reverse the subarray containing the first 2 elements of the array. Array becomes: `[2 4 7 8]`. (*Array is completely sorted now*).

Now, you are required to write an **iterative function** to sort a given array of integers using the Pancake Sort algorithm described above. Of course, you will need the two recursive functions (i.e. function **`findLargestIndex`** from **Task # 0**, and function **`reverseSubArray`** from **Task # 1**). The prototype of your function should be:

```
void pancakeSortIter (int* arr, int start, int end)
```

Task # 3

Proceed as in **Task # 2**, but now the Pancake sorting function should be implemented as a **recursive** function. The prototype of your function should be:

```
void pancakeSortRec (int* arr, int start, int end)
```

Task # 4

Implement a **recursive** C++ function which takes an array of integers (containing **n** distinct integers) and another integer **k**, and determines **whether there exist *exactly* two elements in the array whose sum is k.** Your function should return **true** if there exist two elements in the array whose sum is **k**, and it should return **false** otherwise. The prototype of your function should be:

```
bool checkSum (int* arr, int start, int end, int k)
```

Here, **start** and **end** are starting and ending indices of array **arr**. If the array **arr** contains **n** integers, then the initial function call will be **checksum(arr,0,n-1,k)**, where **k** is the desired sum.

For example, if the array contains **{ 5, 8, 2, 7, 3 }** and **k** is **11**, then your function should return **true**, because there exist two elements (**8** and **3**) which sum to **11**. However, if **k** is **6**, then your function should return **false**, because there do not exist two elements in the above array which sum to **6**.

Note: The implementation of the above function MUST be recursive. There should NOT be any loop in your function.

Hints:

- When looking at an element of the array **arr[i]**, what other number do you need to make the sum equal to **k**?
- You will need the implementation of the **recursive Linear Search** function.

Also implement a driver main function to test your implementation. A sample run of your program would look like this (text shown in **Red** is entered by the user):

```
Enter the size of array: 5
Enter the 5 elements of the array: 5 8 2 7 3

Enter k: 11
Yes, 11 can be obtained by adding two elements of the array.

Continue (y/n)? y

Enter k: 6
No, 6 can NOT be obtained by adding two elements of the array.

Continue (y/n)? n
```