Data Structures (Spring-2023)

Deadline: 17th September, 11:59pm

Assignment# 01

Instructions:

- All the submissions will be done on Google classroom.
- You have to submit cpp file named (21I-XXXX.zip). Naming convention has to be followed strictly. 30% marks will be deducted for not following submission guidelines.
- The student is solely responsible to check the final zip files for issues like corrupt files, viruses in the file, mistakenly exe sent.
- Be prepared for viva or anything else after the submission of assignment.
- Zero marks will be awarded to the students involved in plagiarism.
- Late submission of your solution is not allowed. Submission within 30 minutes after deadline will be accepted with 50% deduction. After that no submission will be accepted.
- Do not use Vectors in this Assignment. Usage of vector will result in straight zero.
- Make sure you submit the correct code because test cases will be run on them.
- No part of code should be commented.
- Understanding the assignment is also part of assignment.
- All classes made must be templatized.

- **You need to combine all work in single cpp file and submit.**

Problem 1: Arrays: (25 Marks)

Part A:

Implement the class ArrayBasics, which offers basic operations over one-dimensional and two-dimensional arrays. All methods must be implemented as class methods (i.e., static methods). The given prototypes are given in int form. You have to make it in template form to support int, float and double datatypes. The signature of the methods in the ArrayBasics class are the following:

1. ***public static int findMax(int[] A, int i, int j)****:* returns the maximum value occurring in the array A between position i and j.
2. ***public static int findMaxPos(int[] A, int i, int j):*** returns the position of the maximum value in the array A between position i and j.
3. ***public static int findMin(int[] A, int i, int j):*** returns the minimum value in the array A between position i and j.
4. ***public static int findMinPos(int[] A, int i, int j):*** return the position of the minimum value in the array A between position i and j.
5. ***public static void swap(int[] A, int i, int j):*** swaps the elements in position i and j in the array A.
6. ***public static void shiftRight(int[] A, int i, int j)*****:** shifts to the right all the elements of the array A starting from position i and until position j (i.e., moves the element in position k to position k + 1 for all $i \leq k < j$, and leaves position i unchanged).
7. ***public static void shiftLeft(int[] A, int i, int j):*** shifts to the left all the elements of the array A, from position j down to position i (i.e., moves the element in position k to position k - 1 for all $i < k \leq j$, and leaves the position j unchanged).
8. ***public static int[] createRandomArray(int size, int min, int max):*** creates and returns an array of size size, of random elements with values between min and max.
9. ***public static int[][] createRandomMatrix(int rows,int cols, int min, int max):*** creates and returns a two-dimensional array with ***rows*** rows and ***cols*** columns of random elements with values between min and max.
10. ***public static int[] copyArray(int[] A, int size)*****:** returns an array that is a copy of A.
11. ***public static int[][] copyMatrix(int[][] A, int rows, int cols)*****:** returns a two-dimensional array that is a copy of A.
12. ***public static int findInArray(int[] A, int q, int size):*** returns the position of the number q in the array A (returns -1 if q is not present in A).
13. ***public static int findInSortedArrary(int[] A, int q, int size)****:* returns the position of the number q in the sorted array A (returns -1 if q is not present in A). The method assumes that the array A is sorted, it need not be correct if A is not sorted. Exploit the fact that the array is sorted to find an efficient algorithm.
14. ***public static int findFirstInSortedArrary(int[] A, int q, int size):*** returns the first position where the number q occurs in the sorted array A (returns -1 if q is not present in A). As before, the method assumes that the array A is sorted and need not be correct if A is not sorted. Again, exploit the fact that the array is sorted to find an efficient algorithm.

Part B:
Running Time Comparison—Maxsort

Add to your class ArrayUtility two static methods implementing the algorithm Maxsort, that takes an unsorted array of integer numbers as input and sorts it in descending order, by repeatedly doing the following:

- First, it searches in the whole array for the greatest element.
- it then puts this element to the beginning of the array.
- Then, it searches the whole array excluding the first element for the greatest value and puts it to the second position.

Implement the algorithm according to two different strategies:

- By using the method shiftRight(int[] A, int i, int j): if the maximum element is found in position j and needs to be put into position i, then (i) shift A to the right, starting from position i, while remembering the element in position j that will be overridden; (ii) copy the remembered element to position i.
- By using the method swap(int[] A, int i, int j): if the maximum element is found in position i and needs to be put into position j, then use swap to exchange the element in position i with the element in position j.

The perform tests to find out which of the two implementations is faster. Is there an array size for which the running times crosse over? (A size N would be such a cross-over point if for inputs of size less than N, the running times of one algorithm are better, while for inputs of size greater than N, the running times of the other algorithm are better.) To perform your measurements, test using steps:

- creates random arrays of size n = 10, 100, 1000, etc., and
- for each array created, sorts it using the two implementations of Maxsort and measures the running times.

Problem 2: Warehouse Problem (25 Marks)

Micheal Scarn is a forklift operator at Munder Difflin paper company's central warehouse. He needs to ship exactly ***r*** reams of paper to a customer. In the warehouse are n boxes of paper, each one foot in width, lined up side-by-side covering an ***n***-foot wall. Each box contains a known positive integer number of reams, where no two boxes contain the same number of reams. Let $\boldsymbol{B = (b_0,...b_{n-1})}$ be the number of reams per box, where box i located i feet from the left end of the wall contains b_i reams of paper, where $\boldsymbol{b_i != b_j}$ for all $\boldsymbol{i != j}$. To minimize his effort, Scarn wants to know whether there is a *close* pair $(\boldsymbol{b_i,b_j})$ of boxes, meaning that $\boldsymbol{|i-j| < n/10}$, that will *fulfill* order r, meaning that $\boldsymbol{b_i + b_j = r.}$

(a) Given B and r, write an expected O(n)-time algorithm in C++ to determine whether B contains a close pair that fulfills order r.

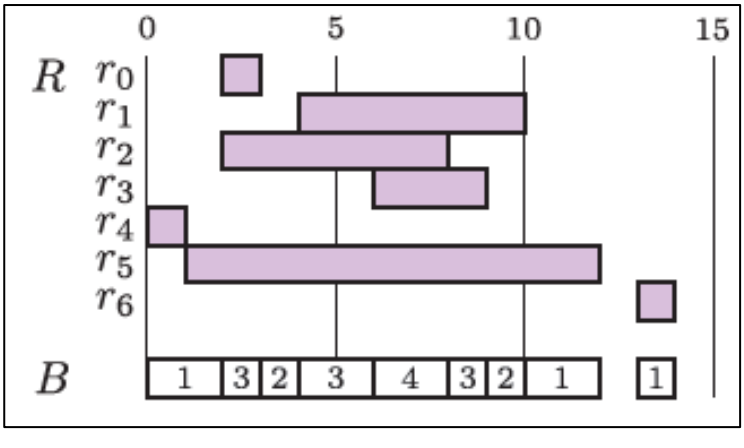
(b) Now suppose that $r<n^2$. Describe a worst-case O(n)-time algorithm to determine whether B contains a close pair that fulfills order r.

Problem 3: Well, Well Well!!! (25 Marks)

The oil wells of tycoon Ron Weaselton will produce m oil barrels this month. Ron has a list of n orders from potential buyers, where the i^{th} order states a willingness to buy a_i barrels for a total price of p_i (not per barrel). Each order must be filled in completely or not at all, and can only be filled once. Ron does not have to sell all of his oil, but he must pay s dollars per unsold barrel in storage costs. Describe an O(nm)-time algorithm to determine which orders to fill so that Ron can maximize his profit.
Function: int maximizeProfit(int m, int n, int a[], int p[], int s);

Problem 4: Booking Predicament: (25 Marks)

Tony is arranging Tim Talks, a lecture series that allows anyone in the university to schedule a time to talk publicly. A talk request is a tuple (s, t), where s and t are the starting and ending times of the talk respectively with s < t (times are positive integers representing the number of time units since some fixed time).
Tim must make room reservations to hold the talks. A room booking is a triple (k, s, t), corresponding to reserving k > 0 rooms between the times s and t where s < t. Two room bookings (k1, s1, t1) and (k2, s2, t2) are disjoint if either t1 ≤ s2 or t2 ≤ s1, and adjacent if either t1 = s2 or t2 = s1. A booking schedule is an ordered tuple of room bookings where: every pair of room bookings from the schedule are disjoint, room bookings appear with increasing starting time in the sequence, and every adjacent pair of room bookings reserves a different number of rooms.
Given a set R of talk requests, there is a unique booking schedule B that satisfies the requests, i.e., the schedule books exactly enough rooms to host all the talks. For example, given a set of talk requests R = {(2, 3), (4, 10), (2, 8), (6, 9), (0, 1), (1, 12), (13, 14)} pictured below, the satisfying room booking is: B = ((1, 0, 2), (3, 2, 3), (2, 3, 4), (3, 4, 6), (4, 6, 8), (3, 8, 9), (2, 9, 10), (1, 10, 12), (1, 13, 14)).



- Given two booking schedules B_1 and B_2, where $n = |B_1| + |B_2|$ and B_1 and B_2 are the respective booking schedules of two sets of talk requests R_1 and R_2, design an O(n)-time algorithm to compute a booking schedule B for $R = R_1 \cup R_2$.
- Given a set R of n talk requests, design an O(n log n)-time algorithm to return the booking schedule that satisfies R.

----------Happy Coding! 😊 ----------