CS213M: Assignment 2

General Instructions and Tips:

- For each problem, test cases (p<x>_t<y>.txt) will be provided in the test folder. Note that your code will be tested on hidden test cases on submission.
- Use the following command to compile p1.cpp: `g++ p1.cpp`. An executable a.out is created. To run the executable, use the command `./a.out`
- USE ‘cin’ and ‘cout’ for taking input and printing respectively. To take input from a file abc.txt, use `./a.out < abc.txt`
- Students are expected to adhere to the highest standards of integrity and academic honesty. Acts such as copying in the examinations and sharing code for the programming assignments will be dealt with strictly, in accordance with the institute's procedures and disciplinary actions for academic malpractice.
- You can use well known algorithms, provided you write the code yourselves, such cases will be handled subjectively.
- We will be using the g++ compiler for compiling the code

P1: Balancing Brackets

A bracket is considered to be any one of the following characters: **(**, **)**, **{**, **}**, **[** or **]**.

Two brackets are considered to be a matched pair if an opening bracket (i.e. **(**, **[**, or **{**) occurs to the left of a closing bracket (i.e. **)**, **]**, or **}**) of the exact same type. There are three types of matched pairs of brackets: **[]**, **{}**, and **()**.

A matching pair of brackets is not balanced if the set of brackets it encloses are not matched. For example, **{[(])}** is not balanced because the contents in between **{** and **}** are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, **(** and the pair of parentheses encloses a single, unbalanced closing square bracket, **]**.

By this logic, we say a sequence of brackets is balanced if the following conditions are met:

- It contains no unmatched brackets.
- The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

Given a string of brackets, determine whether the sequence of brackets is balanced. If a string is balanced, output **1**. Otherwise, output **0**.

Input Format:

The first line contains a single integer **n**, the length of the string.
The next line contains a single string **s** of length **n**, a sequence of brackets.

Constraints:

- 1 <= **n** <= 10^3
- All characters in the sequences ∈ { **{**, **}**, **(**, **)**, **[**, **]** }.

Example :

Input	**Output**
{[()]} **{[(])}** **{{[[(())]]}}**	1 0 1

File to be submitted : p1.cpp

P2: Truck Tour

Consider a circle with **N** petrol pumps spread out randomly on the periphery. Petrol pumps are numbered 0 to N-1 (both inclusive). You have two pieces of information corresponding to each of the petrol pump:

1. The amount of petrol that particular petrol pump will give
2. The distance from that petrol pump to the next petrol pump.

Initially, you have a tank of infinite capacity carrying no petrol. You can start the tour at any of the petrol pumps. Calculate the first point from where the truck will be able to complete the circle. Assume that the truck will stop at each of the petrol pumps. The truck will move **one kilometer for each litre** of the petrol.

Input Format:
The first line will contain the value of **N**.
The next **N** lines each will contain a pair of integers separated by a space denoting the amount of petrol that petrol pump will give and the distance between that petrol pump and the next petrol pump respectively.

Constraints:
1 <= **N** <= 10^5
1 <= Amount of petrol, Distance between petrol pumps <= 10^9
Note: use 'long long' instead of 'int' to avoid integer overflow.

Output Format:
An integer which will be the **smallest** index of the petrol pump from which we can start the tour. Output **-1** if such a tour does not exist.

Example:

Input	**Output**
3 1 5 10 3 3 4	1
3 2 3 4 5 10 11	-1
4 5 3 6 8 4 5 8 7	3

File to be submitted : p2.cpp

P3: Smaller Than Thou

Use a stack to implement the following functionality: Given an indexed list of numbers, determine the next smaller element for each number. You need to output the index of the next smaller element. Assume the sequence to be of non-negative numbers.
Formally, for each **i**, find **j>i** such that **a[i]>a[j]** and **(j-i) is minimum**. If no such **j** exists, output **-1**

For instance, assume the following input sequence:
10 1 2 6 5 9 3

The corresponding next-smaller-element for each is,
10 -> 1 (10>1 and the index of element 1 is 1)
1 -> -1 (no smaller element than this in the remaining sequence)
2 -> -1 (no element less than 2 exists subsequently)
6 -> 4 (6>5 and the index of element 5 is 4)
5 -> 6 (5>3)
9 -> 6 (9>3)
3 -> -1 (trivial)

The trivial algorithm to do this can be in two passes over the whole sequence, and will take $O(n^2)$ time. A stack-based procedure can do this in O(n) time,which is the goal for this assignment..

Input Format:
The first line will contain the value of **N**.
The second line will contain the **N** non-negative integers

Constraints:
1 <= **n** <= 10^6

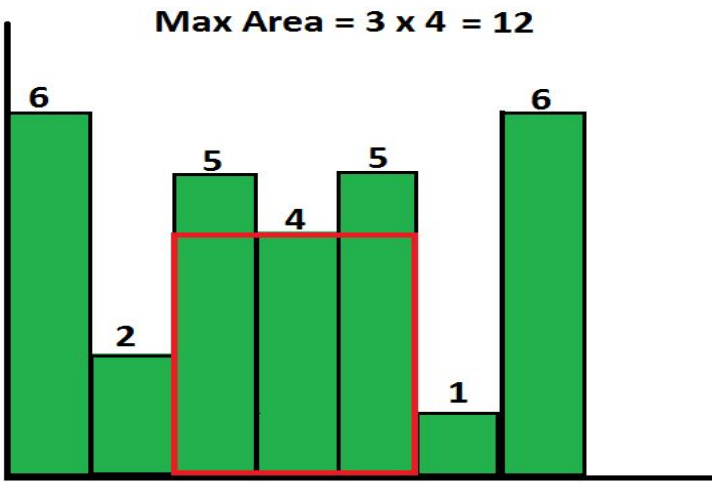
Examples:

Input	**Output**
5 1 4 2 3 7	-1 2 -1 -1 -1
10 1 4 5 13 17 19 8 4 3 0	9 8 7 6 6 6 7 8 9 -1

File to be submitted : p3.cpp
(Tip: You can implement the logic of next smaller element using C++ STL. Read about how to use **stack<T>** where T can be any data type.)

P4: Building area maximization

Consider a sequence of numbers representing the height of buildings adjoining each other. Assume width of the building is 1 unit and we are in 2-D space, so the other dimension is the height.
You need to find a rectangle such that the whole of the rectangle is covered by buildings, and the rectangle has maximum area among all such possible.



Input Format:
The first line will contain the value **N,** the number of buildings.
The second line will contain the **N** non-negative integers, denoting the height of buildings.

Output:
The maximum **rectangular** area covered by the buildings

Examples**:**

Input	**Output**
10 1 4 5 13 17 19 8 4 3 0	39
1 1	1
5 1 4 2 3 7	8

File to be submitted : p4.cpp

P5: Queue using Two Stacks

A queue is an abstract data type that maintains the order in which elements were added to it, allowing the oldest elements to be removed from the front and new elements to be added to the rear. This is called a First-In-First-Out (**FIFO**) data structure because the first element added to the queue (i.e. the one that has been waiting the longest) is always the first one to be removed.

A basic queue has the following operations
Enqueue: add a new element to the end of the queue.
Dequeue: remove the element from the front of the queue and return it.

In this problem, you must first implement a queue using two stacks. Then process queries, where each query is one of the following types:
1 x : Enqueue element **x** at the end of the queue.
2 : Dequeue the element at the front of the queue.
3 : Print the element at the front of the queue. Print **-1** if the queue is empty

Input Format:
The first line contains a single integer **N** denoting the number of queries.
Subsequent **N** lines contains a single query in the form described in the problem statement above. All three queries start with an integer denoting the query, but only query **1** is followed by an additional space-separated value, **x**, denoting the value to be enqueued.

Output Format:
For each query of type **3**, print the value of the element at the front of the queue on a new line.

Example:

Input	**Output**
10 1 42 2 1 14 3 1 28 3 1 60 1 78 2 2 3	14 14 60

Explanation:

We perform the following sequence of 10 actions:

Enqueue 42; queue = {42}.
Dequeue the value at the head of the queue, 42; {}.
Enqueue 14; queue = {14}.
Print the value at the head of the queue, 14; queue = {14}.
Enqueue 28; queue = {14, 28}.
Print the value at the head of the queue, 14 ; queue = {14, 28} .
Enqueue 60 ; queue = {14, 28, 60}.
Enqueue 78; queue = {14, 28, 60, 78}.
Dequeue the value at the head of the queue, 14; queue = {28, 60, 78}.
Dequeue the value at the head of the queue, 28; queue = {60, 78}.
Print the value at the head of the queue, 60 ; queue = {60, 78}.

File to be submitted : p5.cpp

Submission Guidelines:

Please adhere strictly to the following submission format. If you are not able to solve all the problems, DO NOT edit the provided file in order to help evaluate your submission. Your submission should have the following directory structure:

<roll-number>_A2

|----P1

|-------p1.cpp

|----P2

|-------p2.cpp

|----P3

|-------p3.cpp

|----P4

|-------p4.cpp

|----P5

|-------p5.cpp

Zip the folder, and name the zip file **<roll_number>_A2.zip**.