# CS213M: Assignment 4

## 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 in 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 : Flip the bits

There are **n** integers $a_1, a_2, ..., a_n$. Each of those integers can be either **0** or **1**. You are allowed to do **exactly one move**: choose two indices **i** and **j** ($1 \le i \le j \le n$) and flip all values $a_k$ for which their positions are in range [i, j] (that is $i \le k \le j$). Flipping the value of x means to apply the operation `x = 1 - x`.

The goal is that, after exactly one move, obtain the maximal number of ones.

Input:
The first line of the input contains an integer **n** ($1 \le$ **n** $\le 100$).
In the second line of the input there are n integers: $a_1, a_2, ..., a_n$. It is guaranteed that each of those n values is either 0 or 1.

Output:
Print an integer — the maximal number of 1s that can be obtained after exactly one move.

Examples:

| Input | Output |
|---|---|
| 5<br>1 0 0 1 0 | 4 |
| 4<br>1 0 0 1 | 4 |

In the first case, flip the segment from 2 to 5 ($i = 2, j = 5$). That flip changes the sequence, it becomes: [1 1 1 0 1]. So, it contains four ones. There is no way to make the whole sequence equal to [1 1 1 1 1].

In the second case, flipping only the second and the third element ($i = 2, j = 3$) will turn all numbers into 1.

*File to be submitted : p1.cpp*

# P2 : Keeping up with the medians

Consider that integers are read from a data stream. You need to find median of elements read so for in an efficient way. Assume that there will be no duplicates in the stream.
We will simulate the stream and the query mechanism by defining two operations, explained in the input section.

A trivial algorithm to solve this problem is by implementing Insertion sort, thereby maintaining a sorted array to serve the query for the median. If we just ignore the stream, then this approach will have a complexity of $O(n^2)$. But can we do better?
Note that implementing algorithms for such dynamic environments are classified as **online algorithms.** If we have a slow algorithm, such as Insertion sort, to process the stream, then the algorithm in itself becomes a bottleneck for further processing of the stream. So can we find the medians in O(*nlog(n)*)?

Input:

The first line will contain the number of queries, **q**.
The subsequent **q** lines will have the following two type of queries:

1. **c** *x* : Consume integer *x.*
2. **m** : Print the median so far

Assume that the very first query will not be **m.**

Output:
Print the median encountered so far whenever **m** is encountered

Example:

| Input | Output |
|---|---|
| 7<br>c 5<br>m<br>c 15<br>c 1<br>m<br>c 3<br>m | 5<br>5<br>4 |

Initially 5 is the only element consumed, hence the median is 5. After 15 and 1 are consumed, the median remains the same. On the consumption of 3, the median becomes the average of 5 and 3, which is 4

*File to be submitted : p2.cpp*

# P3 : Who are your Ancestors?

Implement the Binary Search Tree. You can have a look at the Wikipedia of the same : https://en.wikipedia.org/wiki/Binary_search_tree.

Use a node class for building the tree. **Revise the concept of pointers and classes before attempting this question.** A typical node structure that can be used is as follows :

```
class Node{
    public : int value;
        Node* leftChild;
        Node* rightChild;
}
```

You can add additional member variables and member functions depending on the use case that follows. Also you could have used a struct instead of a class for the same application.

You will be given **n** positive integers. Using the **n** values you have to construct a binary search tree. The insertions in the binary search tree are done in the order of occurrence.

Using the constructed Binary Search Tree and given a query node, write a program that prints all the ancestors of the query node in the given binary tree.

A node **x** is an ancestor of node **y**, iff there is a downstream path from **x** to **y.**

Input **:**
**n** : number of nodes in the tree.
**n** space separated **distinct** integers corresponding to the values of each node.
**queryNode** : the node whose ancestors are to be printed

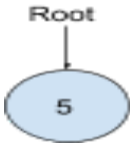
Output**:**
n space separated nodes in increasing order of ancestry starting from the queryNode(excluding) to the root.
If the queryNode is the root node or it is not present in the tree, then print **-1**.

**Worked out Example:**

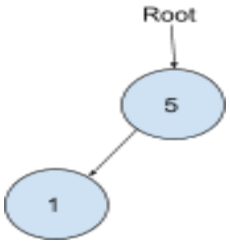
Input : n = 4
5 1 3 7
3

Soln :

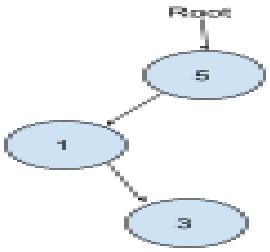
1) First we insert the node with value 5, the BST looks like :



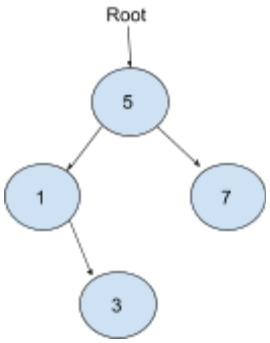

2) Then we insert the value 1, since 1 < 5, it will be the left child of 5 , the tree now looks like:




3) Then we insert the value 3, since 3 < 5, we look at the left subtree and since 3 > 1 we insert it at the right subtree of 3.




4) While inserting 7, since 7 > 5, we insert it at the right subtree of 5, the final tree now is :



Now the query node is that of value 3, looking at the graph we see that the ancestors of 3 are 1 and 5 in order. Therefore the output is : 1 5.

Examples**:**

| **Input** | **Output** |
|---|---|
| 3<br>4 3 6<br>3 | 4 |
| 4<br>1 2 3 4<br>4 | 3 2 1 |
| 6<br>4 5 1 3 6 9<br>7 | -1 |
| 8<br>23 12 8 9 1 89 13 4<br>23 | -1 |

*File to be submitted : p3.cpp*

# P4 : Are you in the loop?

Given a vertex, check whether it belongs to a loop in the graph. A loop or a cycle is a path of edges and vertices wherein a vertex is reachable from itself.
This problem is important in the field of Computer Networks, as we can afford to remove a router in the network if it belongs to a loop (think why?)

Graph : (V, E), where V : set of vertices number 0 to |V| - 1 and E be the set of edges in the form of tuples : ($V_i$, $V_j$), which means there is a directed edge from $V_i$ to $V_j$. There are no self loops in the graphs. I.e. there are no edges of the form ($V_i$, $V_i$).
(Hint : Read about Depth First Search and Breadth First Search in a graph.)

Input :
n, m : n = |V| : number of vertices, m = |E| : the number of edges;
Next m lines of the form $V_i$ $V_j$ which denotes an edge from $V_i$ to $V_j$. i< n, j< n.
q : the given query vertex, q < n, implies we want to check if $V_q$ is in a loop.

Output :
1 : if that vertex is a part of a loop / cycle
0 : otherwise

Examples :

| Input | Output |
|---|---|
| 3 3<br>0 1<br>1 2<br>2 0<br>1 | 1 |
| 3 3<br>0 1<br>1 2<br>0 2<br>1 | 0 |
| 4 5<br>1 2<br>2 1<br>2 3<br>0 2<br>0 1<br>1 | 1 |

*File to be submitted : p4.cpp*

# P5 : Minimum Spanning Tree (Bonus) :

Given a completely-connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, completely-connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. Completely-connected graph means that there exists an edge between every pair of vertices.
Thus, a minimum spanning tree has (|V| – 1) edges where |V| is the number of vertices in the given graph. The graph has $|V|^2$ edges, represented as an adjacency matrix, denoted from now on as **adj**. The adjacency matrix element **adj[i][j]** is the weight of the edge going from vertex i to vertex j, here 0 < i, j < |V| - 1. You might notice that in the undirected case, **adj** is a symmetric matrix.

Input :

**N** : number of vertices in the graph (|V|)

Next **N** lines : each line contains N space separated non negative integers of the adjacency matrix.

(Thus a total of $N^2$ non negative integers)

Output :

Output a single integer, which is the sum of the weights of the edges in the MST.

Examples :

| Input | Output |
|---|---|
| 2<br>5 0<br>0 5 | 0 |
| 3<br>0 5 1<br>5 0 3<br>1 3 0 | 4 |
| 2<br>0 5<br>5 0 | 5 |

*File to be submitted : p5.cpp*