Lab 10 22 Oct

Q1 (30%)

Code to be shared: *Binary Heap*

Give an algorithm to find all nodes greater than some value, X, in a max-heap. Your algorithm should run in O(K), where K is the number of nodes output. You may use the heap data structure discussed in the class and implement the printAllNodesGreaterThan(x) of the MaxHeap class defined in the class or your own 3-ary heap developed in Lab 8.
You must be using an O(N) time method for buildheap.

Weight: Only 10% weight will be given to test cases. The remaining 20% for the use of build heap and O(K) time complexity apart from build heap.
Constraints:
1<=N<=10^5
Values in the array in the range [- 10^7, 10^7]

STL: List, Stack,Queue and Vector any of these can be used. No other STL usage is allowed.

Input Format

Line 1: The number of elements in the heap N
Line 2: Value of X
Lines 3 to 2+N: Values in the heap
(**Note:** The values are not in any particular order. You have to use the buildheap method to insert these values into the heap)

Output Format
List of elements greater than X in the ascending order of values.

Input1
10
74
25
90
65

35
87
120
75
65
12
39

Output1:

75
87
90
120

Q2 (35%)

This assignment is to implement an algorithm to test if a given input undirected graph is 2-edge connected or not.

You can use the graph class you have built in Lab 9. Implement a new member function TwoEdgeConn() that returns true if the graph is 2-edge connected and false otherwise. The function TwoEdgeConn() should implement the recursive DFS based algorithm 2EC discussed in the class. See the figure below for a reference.

Algorithm: 2-edge connectedness

Algorithm 14.8: int $\mathrm{2EC}$(Graph G, vertex v)

```
v.visited := True;
v.arrival := time++;
deepest := v.arrival;
for w ∈ G.adjacent(v)−{v.parent} do
    if w.visited == False then
        w.parent := v;
        deepest' := 2EC(G, w);
    else
        deepest' := w.arrival;
    deepest := min(deepest, deepest');
v.departure := time++;
if v.parent ≠ Null and v.arrival == deepest then raise "Bridge found!";
return deepest;
```

STL: List, Stack,Queue and Vector any of these can be used. No other STL usage is allowed.

Input Format
First line of input contains non negative integer n the number of vertices and second line contains non negative integer m the number of edges. Each of the next m lines denotes pairs of vertices representing edges in the graph.

Constraints
1<=n<=10^5
1<=m<=10^6

Output Format
output either prints “yes” if the graph is 2-Edge connected, else prints “no”.

Example1:

Input1

5

8

0 1

0 2

1 3

0 4

3 4

2 4

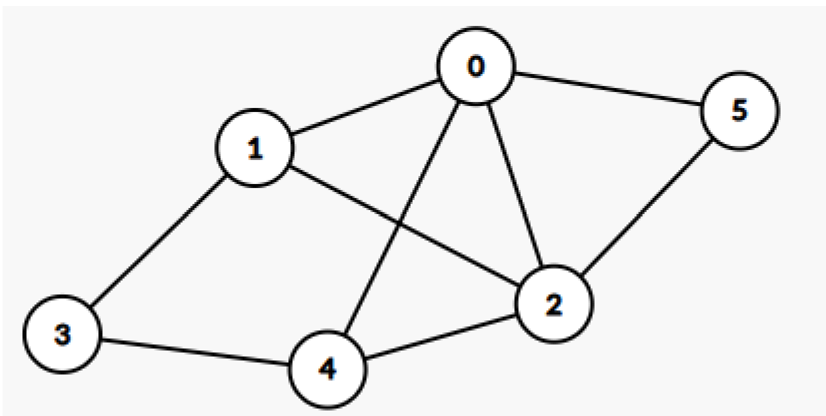
0 5

1 2

5 2

Output1

yes



Example 2:

Input2

6

8

0 1

0 2

1 3

1 4

3 4

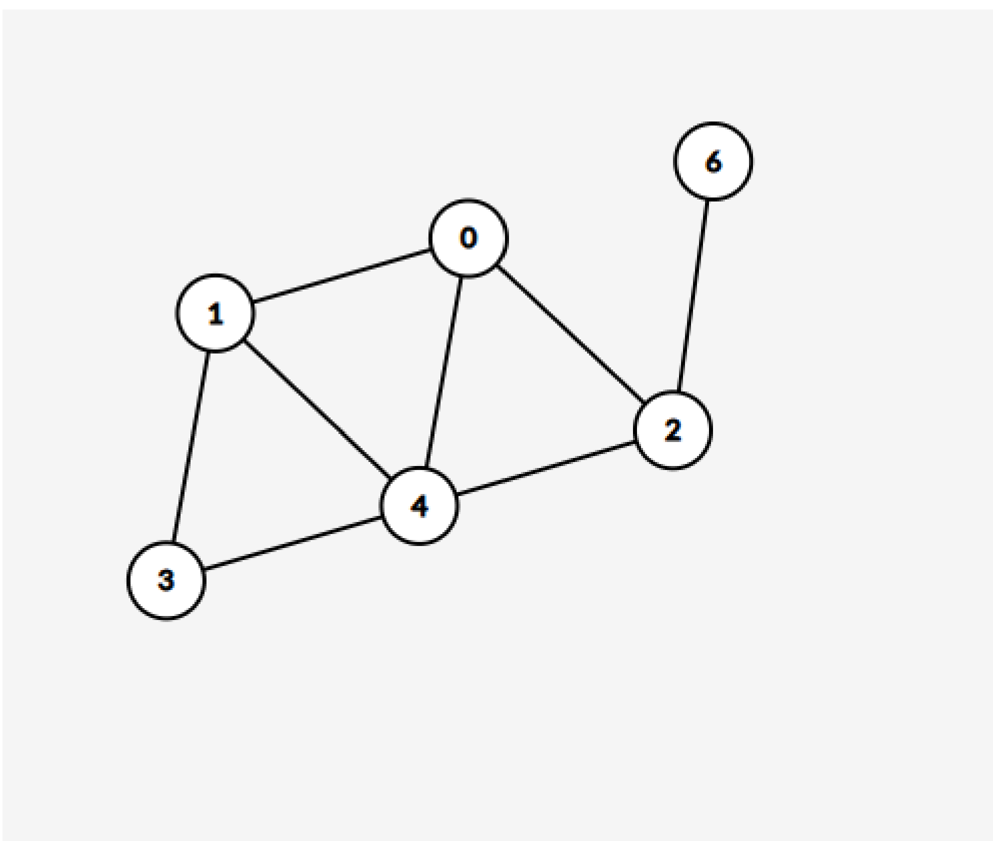
2 4

0 4

2 6

Output2

no



(Bonus 2 points) Modify your program to print the list of all bridge edges in the graph. Submit this in a separate file, no test cases will be provided.

Q3 (35%)

We are given a set of tasks along with dependencies between them. The dependencies are represented as a directed graph. Your task is implement an algorithm that assigns a sequence numbering (1 to n) to each of the tasks such that:

For every task v,

If v not dependent on only other task, then seq(v) =1;

Else

seq(v) = the smallest number i such that there is a task u with seq(u) = i-1 such that v is dependent on u and for any task w that v is dependent on w, we have seq(w)<= i-1.

Complexity: O(m+n), where n is the number of tasks and m is the number of dependencies.

STL usage: You may use stack, queue, list and vector. For graphs you need to use the graph class developed in the previous lab. No other STL usage is allowed.

Input Format

First line of input contains non negative integer n the number of tasks and second line contains non negative integer m the number of dependencies. Each of the next m lines denotes pairs of vertices representing dependencies between two tasks.

Constraints

1<=n<=10^5

1<=m<=10^6

Output Format

If there is a cycle of dependencies among tasks then output “Cycle”.

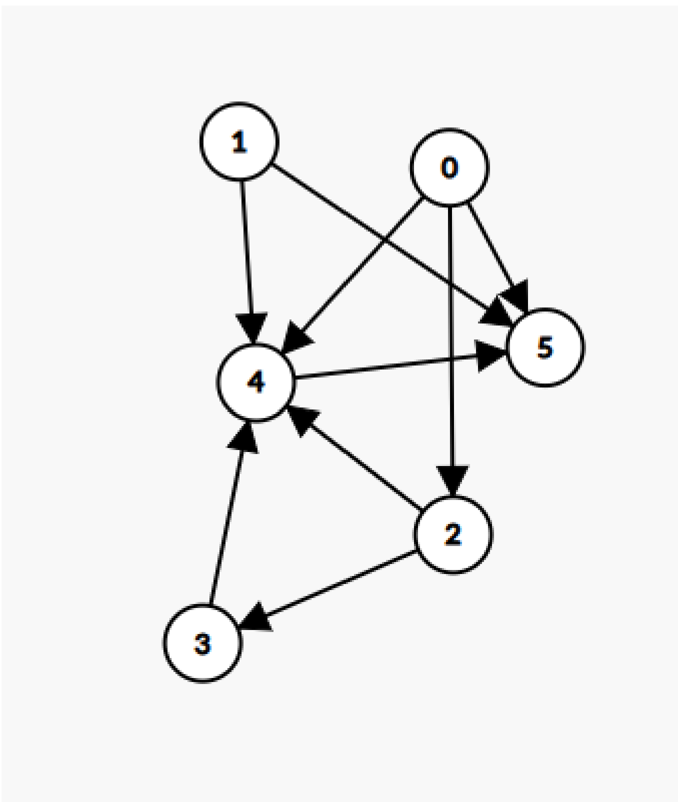
Otherwise, there should be n lines of output and each line should contain the pair v seq(v), sorted in the ascending order of vertex indices, i.e. 0 to n-1.

Example1:

Input 1:

6
9
0 2
0 4
0 5
1 4
1 5

2 3
2 4
4 5
3 4
Output1:
0 1
1 1
2 2
3 3
4 4
5 5



Example2:
Input 2:
9
13
0 2
0 4
0 5
1 4
1 5
2 3
2 4

7 5
3 4
1 7
0 8
2 6
3 6

Output 2
0 1
1 1
2 2
3 3
4 4
5 3
6 4
7 2
8 2