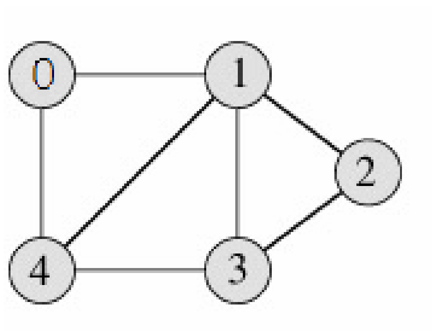
EXERCISE 4
Implementation of Graphs

Aim: To develop a program to demonstrate operations on Graphs.

Description: A Graph G=(V,E) consists of a finite non empty set of vertices V also called points or nodes and a finite set E of unordered pairs of distinct vertices called edges or arcs or links. Following two are the most commonly used representations of graph.

- Sequential Representation or Adjacency Matrix
- Linked Representation or Adjacency List

1. Adjacency Matrix
Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. The adjacency matrix example graph is shown below.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figure: Adjacency Matrix Representation

2. Adjacency List
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the i'th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

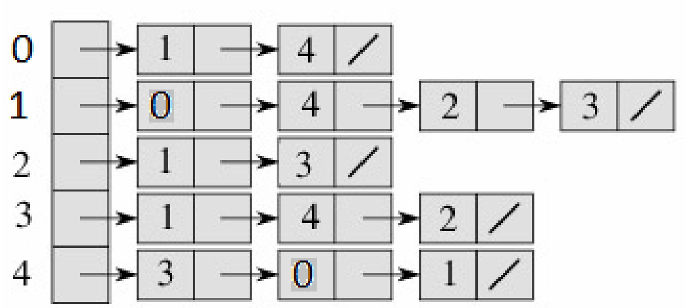


Figure: Adjacency List Representation

Operations on Graphs:

The commonly performed operations on the Graph are:

1. Creating/Storing a graph
2. Inserting a vertex
3. Deleting a vertex
4. Inserting an edge
5. Deleting an edge
6. Traversal of graph

A Graph can be traversed in two ways:

- **Depth-first search (DFS)** is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

- **Breadth-first search (BFS)** is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and explores the neighbour nodes first, before moving to the next level neighbours.

Algorithm for Breadth-first search (BFS):

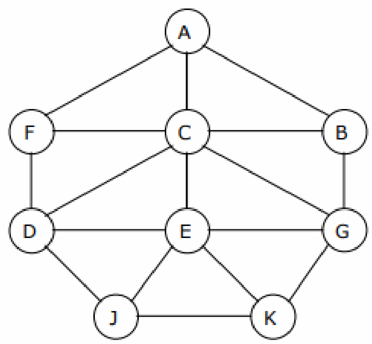
Procedure BFT (s)

```
/* s is the start vertex of the traversal in an undirected graph G */
/* Q is a queue which keeps track of the vertices whose adjacent nodes are to be visited */
/* Vertices which have been visited have their 'visited' flags set to 1 (i.e.) visited (vertex) = 1.
   Initially, visited (vertex) = 0 for all vertices of graph G */
{
        Initialize queue Q;
        visited(s) = 1;
        call ENQUEUE (Q,s);

        while not EMPTY_QUEUE(Q) do
        call DEQUEUE (Q,s)
        Print(s);

        for all vertices v adjacent to s do
                if (visited (v) = 0) then
                {
                call ENQUEUE (Q, v);
                visited (V) =1;
                }
                end
        end while
}
```

Example: Consider the following graph –



A Graph G

Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth First Search Traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Algorithm for Depth-first search (DFS):

Procedure DFT(s)

```
/* s is the start vertex */
        visited(s) = 1;
        Print (s); /* Output visited vertex */
        for each vertex v adjacent to s do
                if visited(v) = 0 then
                    call DFT(v);
        end
end DFT.
```

Depth First Search Traversal:

The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

Program:

```
#include <iostream>
#include <vector>
#include <stack>
#include <queue>
using namespace std;

class Graph
{
private:
    vector<vector<int>> adj;
public:
    Graph(int v) {
        adj = vector<vector<int>> (v+1, vector<int>(v+1, 0) );
    }

    void addEdge(int u, int v)
    {
        adj[u][v] = 1;
        adj[v][u] = 1;
    }

    void delEdge(int u, int v)
    {
        adj[u][v] = 0;
        adj[v][u] = 0;
    }
```

```
void adjMatx()
{
    for(int i=1; i<adj.size(); i++)
    {
        for(int j=1; j<adj[i].size(); j++)
            cout << adj[i][j] << "  ";
        cout << "\n";
    }
}

void delVtx(int v)
{
    if(v < adj.size())
    {
        // To Delete Vertex Row
        adj.erase(adj.begin()+v);

        //To Delete Vertex Column, iterate through each row
        // and delete the vertex index
        for(int i=0; i<adj.size(); i++)
        {
            if(v < adj[i].size())
                adj[i].erase(adj[i].begin()+v);
        }
    }

}

void BFS(int vtx)
{
    queue<int> Q;
    vector<int> visited(adj[1].size(), 0);

    cout << "Breadth First Traversal of the Graph: ";
    cout << vtx << " ";
    visited[vtx] = 1;
    Q.push(vtx);
    while(!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for(int v=1; v<=adj[u].size(); v++)
        {
            if(adj[u][v]==1 && visited[v]==0)
            {
                cout << v << " ";
                visited[v] = 1;
                Q.push(v);
```

```
            }
        }
    }
    cout << endl;
}

void DFS(int vtx)
{
    stack<int> st;
    vector<int> visited(adj[1].size(), 0);

    cout << "Depth First Traversal of the Graph: ";
    cout << vtx << " ";
    visited[vtx] = 1;
    st.push(vtx);
    while(!st.empty())
    {
        int u = st.top();
        st.pop();
        for(int v=adj[u].size(); v>=0; v--)
        {
            if(adj[u][v]==1 && visited[v]==0)
            {
                cout << v << " ";
                visited[v] = 1;
                st.push(v);
            }
        }
    }
    cout << endl;
}
};

int main()
{
    int n, opt, i, j;
    cout <<"---- GRAPH ----\n";
    cout << "Enter No. of Vertices: ";
    cin >> n;

    Graph g(n);

    do
    {
        cout << "\n--- OPERATIONS ---";
        cout << "\n[1] Add Edge";
        cout << "\n[2] Delete Edge";
        cout << "\n[3] Delete Vertex";
        cout << "\n[4] Breadth First Search (BFS)";
```

```
        cout << "\n[5] Depth First Search (DFS)";
        cout << "\n[6] Display Adjacency Matrix";
        cout << "\n[7] Exit";
        cout << "\nEnter Your Choice: ";
        cin >> opt;

        switch(opt)
        {
        case 1:
            cout << "Enter Vertices to Add Edge: ";
            cin >> i >> j;
            g.addEdge(i,j);
            break;
        case 2:
            cout << "Enter Vertices to Delete Edge: ";
            cin >> i;
            g.delEdge(i,j);
            break;
        case 3:
            cout << "Graph After Deleting Last Vertex: ";
            g.delVtx(i);
            break;
        case 4:
            cout << "Enter Starting Vertex: ";
            cin >> i;
            g.BFS(i);
            break;
        case 5:
            cout << "Enter Starting Vertex: ";
            cin >> i;
            g.DFS(i);
            break;
        case 6:
            cout << "---- Adjacency Matrix ---\n";
            g.adjMatx();
            break;
        }

    } while(opt!=7);
}
```

Output:

```
---- GRAPH ----
Enter No. of Vertices: 5

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 1
Enter No. of Edges: 4
Enter Vertices Set to Add Edges: 1 2  1 5  2 3  4 3

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 2
Enter Vertices to Delete Edge: 1 5

Graph after Edge Deletion:
0  1  0  0  0
1  0  1  0  0
0  1  0  1  0
0  0  1  0  0
0  0  0  0  0

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
```

```
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 1
Enter No. of Edges: 2
Enter Vertices Set to Add Edges: 2 5  2 4

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 4
Enter Starting Vertex: 4
Breadth First Traversal of the Graph: 4 2 3 1 5

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 5
Enter Starting Vertex: 4
Depth First Traversal of the Graph: 4 3 2 5 1

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 6
```

```
---- Adjacency Matrix ---
0  1  0  0  0
1  0  1  1  1
0  1  0  1  0
0  1  1  0  0
0  1  0  0  0

--- OPERATIONS ---
[1] Add Edge
[2] Delete Edge
[3] Delete Vertex
[4] Breadth First Search (BFS)
[5] Depth First Search (DFS)
[6] Display Adjacency Matrix
[7] Exit
Enter Your Choice: 7
```

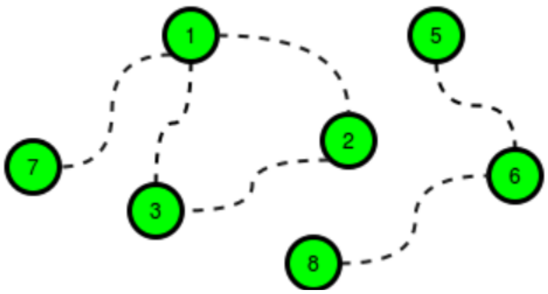
Applications of Graphs

Determine the minimum cost to provide library access to all citizens of HackerLand. There are n cities numbered from 1 to n. Currently there are no libraries and the cities are not connected. Bidirectional roads may be built between any city pair listed in $cities$. A citizen has access to a library if:

- Their city contains a library.
- They can travel by road from their city to a city containing a library.

Example

The following figure is a sample map of HackerLand where the dotted lines denote possible roads:



$c\_road = 2$

$c\_lib = 3$

$cities = [[1, 7], [1, 3], [1, 2], [2, 3], [5, 6], [6, 8]]$

The cost of building any road is $cc\_road = 2$, and the cost to build a library in any city is $c\_lib = 3$. Build 5 roads at a cost of $5 \times 2 = 10$ and 2 libraries for a cost of 6. One of the available roads in the cycle $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is not necessary.

There are q queries, where each query consists of a map of HackerLand and value of c_lib and c_road. For each query, find the minimum cost to make libraries accessible to all the citizens.

Function Description

Complete the function roadsAndLibraries in the editor below.

roadsAndLibraries has the following parameters:

- int n: integer, the number of cities
- int c_lib: integer, the cost to build a library
- int c_road: integer, the cost to repair a road
- int cities[m][2]: each $cities[i]$ contains two integers that represent cities that can be connected by a new road

Returns

- int: the minimal cost

Program:

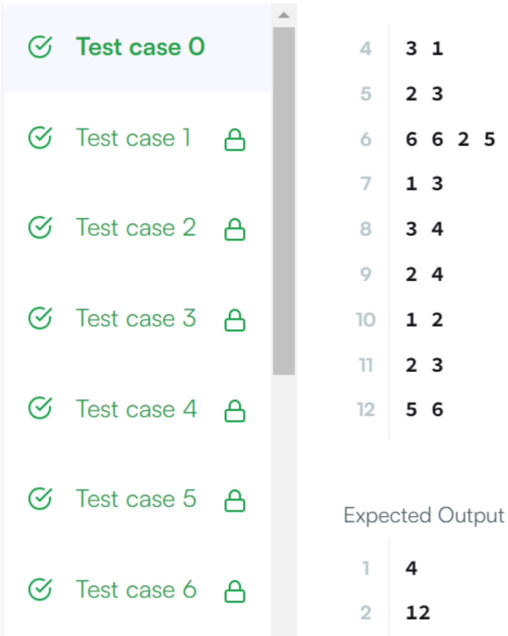
```
void bfs(int start, unordered_map<int, vector<int>>& adjList, unordered_set<int>& vis) {
    queue<int> pq;
    pq.push(start);
    vis.insert(start);
    while (!pq.empty()) {
        auto node = pq.front();
        pq.pop();
        for (auto nbs: adjList[node]) {
            if (vis.find(nbs) == vis.end()) {
                vis.insert(nbs);
                pq.push(nbs);
            }
        }
    }
}
long roadsAndLibraries(int n, int c_lib, int c_road, vector<vector<int>> cities)
{
    if (c_lib <= c_road)
            return static_cast<long>(n)*c_lib;

    unordered_map<int, vector<int>> adjList;
    // build adjList
    for(auto ct: cities) {
        adjList[ct[0]].push_back(ct[1]);
        adjList[ct[1]].push_back(ct[0]);
    }
    // unvisited list
    unordered_set<int> vis;
    long cost = 0;
    int region = 0;

    for (int i=1; i<= n ;i++) {
        if (vis.find(i) == vis.end()) {
            cost += static_cast<long>(c_lib);
            region++;
            // find out all connected path to current cty
            bfs(i,adjList, vis);
        }
    }

    // calculate final cost
    // total path = (cities -1) - (region -1) = cities - region
    cost += (vis.size() - region) * static_cast<long>(c_road);
    return cost;
}
```

Output:



2. Special Edge – 3 (Hacker Earth)

Program:

```
#include <bits/stdc++.h>
using namespace std;
const int max_size = 1e5+1;
int M, N, T, a, b;
int         A[max_size];
int         g[max_size];
int         h[max_size];
pair<int,int> E[max_size];
vector<int> adj[max_size];
bool    visited[max_size];
int64_t max_strength;
 inline auto next_int() { int num; cin >> num; return num; }
 inline auto next_ipair() {
    const auto u = next_int(), v = next_int();
    return pair<int,int>(u,v); }
```

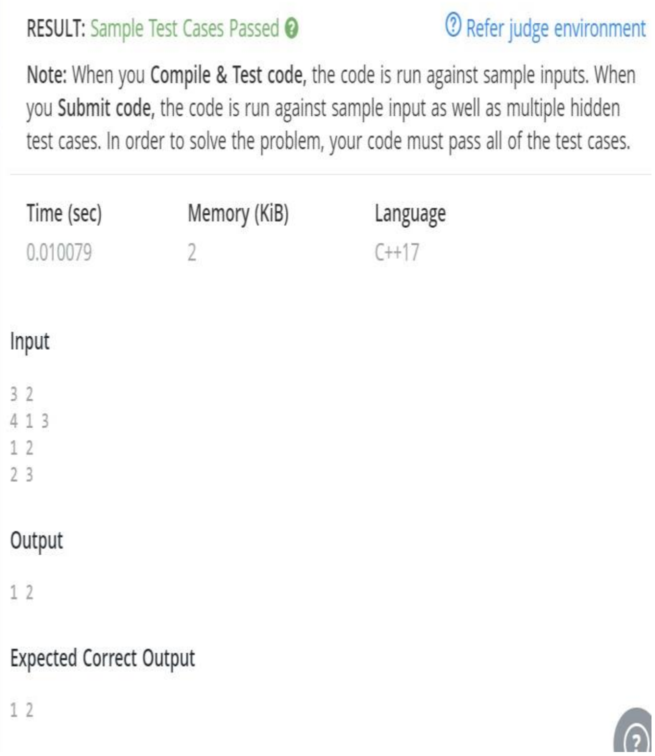
```
inline auto bridge(int x, int y) {
  const int t = A[y], u = min(x,y), v = max(x,y);
  const int64_t strength = 1ll*t*(T-t);
  if (strength > max_strength or
     (strength == max_strength and
     (u < a or (u == a and v < b))))
     max_strength = strength, a = u, b = v; }
void dfs(int x, int p) {
  static int time = 0;
  g[x] = h[x] = ++time, visited[x] = true;
  for (auto i: adj[x]) {
     const auto [u,v] = E[i];
     const auto y =  x == u ? v: u;
     if (y != p) {
        if (visited[y])
           h[x] = min(h[x],g[y]);
        else if (dfs(y,x), h[x] = min(h[x],h[y]), A[x] += A[y], g[x] < h[y])
           bridge(x,y); } } }
int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  N = next_int(), M = next_int(), a = b = N+1;
  generate(A+1,A+a,next_int);
  generate(E,E+M,next_ipair);
  T = accumulate(A+1,A+a,0);
  for (int i = 0; i < M; ++i) {
     const auto [u,v] = E[i];
     adj[u].push_back(i),
     adj[v].push_back(i); }
  dfs(1,0);
  cout << a << ' ' << b;
}
```

Output:

RESULT: Accepted Refer judge environment

Score	Time (sec)	Memory (KiB)	Language
0	0.1859	7796	C++17

Input	Result	Time (sec)	Memory (KiB)	Score	Your output	Correct output	Diff
Input #1	Accepted	0.017731	2	10			
Input #2	Accepted	0.025685	6556	10			
Input #3	Accepted	0.025815	6868	10			
Input #4	Accepted	0.010182	2	10			
Input #5	Accepted	0.009534	2	10			
Input #6	Accepted	0.017564	2	10			
Input #7	Accepted	0.025756	6092	10			
Input #8	Accepted	0.033827	7796	10			
Input #9	Accepted	0.009596	2	10			
Input #10	Accepted	0.01021	2	10			



RESULT: Sample Test Cases Passed Refer judge environment

Note: When you **Compile & Test code,** the code is run against sample inputs. When you **Submit code,** the code is run against sample input as well as multiple hidden test cases. In order to solve the problem, your code must pass all of the test cases.

Time (sec)	Memory (KiB)	Language
0.010079	2	C++17

Input

```
3 2
4 1 3
1 2
2 3
```

Output

```
1 2
```

Expected Correct Output

```
1 2
```