

## ICPC Team Reference Material

## Contents

<b>1</b>	<b>Setup</b>	
1.1	Vimrc	1
1.2	Capslock as escape	1
1.3	Compilation	1
<b>2</b>	<b>Graph algorithms</b>	
2.1	Adjacency list representation	2
2.2	Articulation points and bridges	2
2.3	Bellman ford	2
2.4	Bi-connected components	3
2.5	Bi-partite graph	3
2.6	Breadth first search (bfs)	3
2.7	Connected components	4
2.8	Cycle detection (directed graph)	4
2.9	Cycle detection (undirected graph)	4
2.10	Depth first search (dfs)	4
2.11	Dijkstra (dense graph)	5
2.12	Dijkstra (grid)	5
2.13	Dijkstra (negative weighted graph)	5
2.14	Dijkstra (sparse graph)	5
2.15	Directed cyclic graph into acyclic	6
2.16	Edge classification	6
2.17	Eulerian tour tree	7
2.18	Flood fill	7
2.19	Floyd warshall (all pairs shortest path)	7
2.20	Minimum spanning tree (kruskal)	7
2.21	Kth ancestor and lowest common ancestor (binary lifting)	8
2.22	Lowest common ancestor (euler tour)	9
2.23	Minimum vertex cover	9
2.24	Restoring the path	10
2.25	Shortest path faster algorithm (spfa)	10
2.26	Single source shortest path	10
2.27	Single source shortest path (grid)	11
2.28	Tarjan (strongly connected components)	11
2.29	Topological sort (dfs)	11
2.30	Topological sort (kahns algorithm)	11
2.31	Tree diameter	12
2.32	0-1 bfs	12
2.33	0-1 bfs (grid)	12
<b>3</b>	<b>Data structures</b>	
3.1	Union find disjoint sets	13
3.2	Segment tree (rmq)	13
3.3	Merge sort tree	14
3.4	Sparse table (rmq)	15
3.5	Sparse table (rsq)	15
3.6	Merge sort	15
3.7	Selection sort	16
3.8	Bubble sort	16
<b>4</b>	<b>Mathematics</b>	
4.1	Euler totient function	16
4.2	Euler phi (sieve)	17
4.3	Extended wheel factorization	17
4.4	Least prime factorization	17
4.5	Miller rabin test	17
4.6	Mobius function	17
4.7	Mobius (sieve)	18
4.8	Phi factorial	18
4.9	Pisano periodic sequence	18
4.10	Simple sieve	20
4.11	wheel sieve	20
4.12	Linear sieve	21

4.13	Segmented sieve	21
4.14	The stable marriage problem	21
<b>5</b>	<b>String processing</b>	<b>21</b>
5.1	Trie	21
5.2	Knuth Morris Pratt (kmp)	22
<b>6</b>	<b>Geometry</b>	<b>22</b>
6.1	Point	22
<b>7</b>	<b>More advanced topics</b>	<b>23</b>
7.1	A* algorithm	23
7.2	Mo's algorithm	23
7.3	Square root decomposition	24
<b>8</b>	<b>Miscellaneous</b>	<b>24</b>
8.1	C++ template	24
8.2	Double comparison	25
8.3	Fast input/output	25
8.4	Gcd & Lcm	25
8.5	Modular calculations	25
8.6	Debugging tools	25
8.7	Overloaded operators to accept 128 bit integer	26
8.8	Policy based data structures	26
8.9	Pseudo random number generator	26
8.10	Stress test	26

## 1 Setup

### 1.1 Vimrc

```
1 let mapleader = "\"
2 syntax on
3 filetype plugin on
4 set nocompatible
5 set autoread
6 set foldmethod=marker
7 set autoindent
8 set clipboard+=unnamedplus
9 set number relativenumber
10 set shiftwidth=2 softtabstop=2 expandtab
11 map <leader>c :w! && !compile %:pr<CR>
12 vmap < <gv
13 vmap > >gv
```

### 1.2 Capslock as escape

```
1 setxkbmap -layout us
2 xmodmap -e 'clear Lock'
3 xmodmap -e 'keycode 66 = Escape'
```

### 1.3 Compilation

```
1 #!/bin/bash
2 # put this file in .local/bin or add its dir to the PATH variable
3 compile() {
4     g++ -Wall -Wextra -Wshadow -Ofast -std=c++17 -pedantic -Wformat=2 -Wconversion -Wlogical-op -Wshift-
        overflow=2 -Wduplicated-cond -Wfloat-equal -fno-sanitize-recover -fstack-protector -fsanitize=
        address,undefined -fmax-errors=2 -o "$1" {,.cpp}
5 }
6 compile "$1"
```

## 2 Graph algorithms

### 2.1 Adjacency list representation

```

1 template <class T>
2 class Graph {
3 public:
4     vector <int> _head, _next, _to;
5     vector <T> _cost;
6     int edge_number;
7     bool isDirected;
8
9     Graph() = default;
10    Graph(int V, int E, bool isDirec) {
11        isDirected = isDirec;
12        _head.assign(V + 9, 0);
13        _next.assign(isDirected ? E + 9 : E * 2 + 9, 0);
14        _to.assign(isDirected ? E + 9 : E * 2 + 9, 0);
15        // _cost.assign(isDirected ? E + 9 : E * 2 + 9, 0);
16        edge_number = 0;
17    }
18
19    void addEdge(int u, int v, T w = 0) {
20        _next[++edge_number] = _head[u];
21        _to[edge_number] = v;
22        // _cost[edge_number] = w;
23        _head[u] = edge_number;
24    }
25
26    void addBiEdge(int u, int v, int w = 0) {
27        addEdge(u, v, w);
28        addEdge(v, u, w);
29    }
30
31    void dfs(int node) {
32        vis[node] = true;
33        for(int i = _head[node]; i; i = _next[i]) if(!vis[_to[i]]) {
34            dfs(_to[i]);
35        }
36    }
37 };

```

### 2.2 Articulation points and bridges

```

1 const int N = 1e5 + 9, M = 2e6 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2 ll INF = 0x3f3f3f3f3f3f3f3f;
3
4 int Head[N], Next[M], To[M], Cost[M];
5 int Par[N], dfs_num[N], dfs_low[N];
6 int ne, n, m, u, v, w;
7 int root, rootChildren, dfs_timer, bridgeInx;
8 bool Art[N];
9 vector < pair <int, int> > bridges(M);
10
11 void addEdge(int from, int to, int cost = 0) {
12     Next[++ne] = Head[from];
13     Head[from] = ne;
14     Cost[ne] = cost;
15     To[ne] = to;
16 }
17
18 void _clear() {
19     memset(Head, 0, sizeof(Head[0]) * (n + 2));
20     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
21     memset(Par, -1, sizeof(Par[0]) * (n + 2));
22     memset(Art, 0, sizeof(Art[0]) * (n + 2));
23     ne = dfs_timer = bridgeInx = 0;
24 }
25
26 void Tarjan(int node) {
27     dfs_num[node] = dfs_low[node] = ++dfs_timer;
28
29     for(int i = Head[node]; i; i = Next[i]) {
30         if(dfs_num[To[i]] == 0)
31             {
32                 if(node == root) ++rootChildren;
33
34                 Par[To[i]] = node;
35                 Tarjan(To[i]);
36                 dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
37
38                 if(dfs_low[To[i]] >= dfs_num[node])
39                     Art[node] = true;

```

```

40         if(dfs_low[To[i]] > dfs_num[node])
41             bridges[bridgeInx++] = make_pair(node, To[i]);
42     }
43     else if(To[i] != Par[node])
44         dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
45 }
46 }
47
48 int main() {
49     cin >> n >> m;
50     _clear();
51
52     while(m--) {
53         cin >> u >> v;
54         addEdge(u, v);
55         addEdge(v, u);
56     }
57
58     for(int i = 1; i <= n; ++i)
59         if(dfs_num[i] == 0) {
60             root = i;
61             rootChildren = 0;
62             Tarjan(i);
63             Art[root] = (rootChildren > 1);
64         }
65
66     cout << "Art Points :\n";
67     for(int i = 1; i <= n; ++i) if(Art[i])
68         cout << i << " ";
69
70     cout << "\nBridges :\n";
71     for(int i = 0; i < bridgeInx; ++i)
72         cout << bridges[i].first << " - " << bridges[i].second << endl;
73 }
74 }

```

### 2.3 Bellman ford

```

1 // Bellman-Ford Algorithm
2 // In programming contests, the slowness of Bellman Ford and its negative cycle detection feature
3 // causes it to be used only to solve the SSSP problem on small graph
4 // which is not guaranteed to be free from negative weight cycle
5
6 const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
7 ll INF = 0x3f3f3f3f3f3f3f3f;
8
9 int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
10 ll dis[N];
11
12 void addEdge(int from, int to, int cost) {
13     Next[++ne] = Head[from];
14     Head[from] = ne;
15     Cost[ne] = cost;
16     To[ne] = to;
17 }
18
19 void _clear() {
20     memset(Head, 0, sizeof(Head[0]) * (n + 2));
21     ne = 0;
22 }
23
24 bool hasNC() {
25     for(int i = 1; i <= n; ++i)
26         for(int j = Head[i]; j; j = Next[j])
27             if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]])
28                 return true;
29
30     return false;
31 }
32
33 bool Bellman_Ford(int src) {
34     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
35     memset(Par, -1, sizeof(Par[0]) * (n + 2));
36
37     dis[src] = 0;
38     bool newRelaxation = true;
39
40     for(int i = 2; i <= n && newRelaxation; ++i) {
41         newRelaxation = false;
42         for(int i = 1; i <= n; ++i)
43             for(int j = Head[i]; j; j = Next[j])
44                 if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]]) {
45                     dis[To[j]] = dis[i] + Cost[j];
46                     Par[To[j]] = i;
47                     newRelaxation = true;
48                 }
49     }

```

```
50     return hasNC();
51 }
```

## 2.4 Bi-connected components

```
1  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2  const ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], Next[M], To[M];
5  int Par[N], dfs_num[N], dfs_low[N];
6  int ne, n, m, u, v;
7  int root, rootChildren, dfs_timer;
8  int Stack[N], top, ID;
9  bool Art[N];
10 vector < vector <int> > BiCCs(N), BiCCIDs(N);
11
12 void addEdge(int from, int to) {
13     Next[++ne] = Head[from];
14     Head[from] = ne;
15     To[ne] = to;
16 }
17
18 void _clear() {
19     memset(Head, 0, sizeof(Head[0]) * (n + 2));
20     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
21     memset(Par, -1, sizeof(Par[0]) * (n + 2));
22     memset(Art, 0, sizeof(Art[0]) * (n + 2));
23     ne = dfs_timer = top = ID = 0;
24     BiCCs = BiCCIDs = vector < vector <int> > (N);
25 }
26
27 void Tarjan(int node) {
28     dfs_num[node] = dfs_low[node] = ++dfs_timer;
29     Stack[top++] = node;
30
31     for(int i = Head[node]; i; i = Next[i]) {
32         if(dfs_num[To[i]] == 0) {
33             if(node == root) ++rootChildren;
34
35             Par[To[i]] = node;
36             Tarjan(To[i]);
37
38             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
39
40             if(dfs_low[To[i]] >= dfs_num[node]) {
41                 Art[node] = true;
42                 ++ID;
43                 for(int x = -1; x ^ To[i];) {
44                     x = Stack[--top];
45                     BiCCIDs[x].emplace_back(ID);
46                     BiCCs[ID].emplace_back(x);
47                 }
48                 BiCCIDs[node].emplace_back(ID);
49                 BiCCs[ID].emplace_back(node);
50             }
51             else if(To[i] != Par[node])
52                 dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
53         }
54     }
55 }
56
57 int main() {
58     cin >> n >> m;
59     _clear();
60
61     while(m--) {
62         cin >> u >> v;
63         addEdge(u, v);
64         addEdge(v, u);
65     }
66
67     for(int i = 1; i <= n; ++i)
68         if(dfs_num[i] == 0) { // O(n + m)
69             root = i;
70             rootChildren = 0;
71             Tarjan(i);
72             Art[root] = (rootChildren > 1);
73         }
74
75     for(int i = 1; i <= ID; ++i) {
76         cout << "Component : " << i << " contains : ";
77         for(int j = 0; j < (int)BiCCs[i].size(); ++j)
78             cout << BiCCs[i][j] << " \n"[j == BiCCs[i].size() - 1];
79     }
80 }
```

## 2.5 Bi-partite graph

```
1  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
5  ll dis[N];
6  bool color[N], vis[N];
7
8  void addEdge(int from, int to) {
9     Next[++ne] = Head[from];
10    Head[from] = ne;
11    To[ne] = to;
12 }
13
14 bool checkBiPartite(int node, int par = 0) {
15     if(vis[node])
16         return color[par] != color[node];
17
18     color[node] = color[par] ^ 1;
19
20     vis[node] = true;
21     bool ok = true;
22     for(int i = Head[node]; i; i = Next[i])
23         if(To[i] != par)
24             ok &= checkBiPartite(To[i], node);
25
26     return ok;
27 }
28
29 int main() {
30     cin >> n >> m;
31     while(m--) {
32         cin >> u >> v;
33         addEdge(u, v);
34         addEdge(v, u);
35     }
36
37     bool isBiPartite = true;
38     for(int i = 1; i <= n; ++i) if(!vis[i])
39         isBiPartite &= checkBiPartite(i);
40
41     cout << (isBiPartite ? "YES" : "NO") << endl;
42 }
```

## 2.6 Breadth first search (bfs)

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  typedef int64_t ll;
4
5  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
6  ll INF = 0x3f3f3f3f3f3f3f3f;
7
8  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
9  ll dis[N];
10
11 void addEdge(int from, int to, int cost) {
12     Next[++ne] = Head[from];
13     Head[from] = ne;
14     Cost[ne] = cost;
15     To[ne] = to;
16 }
17
18 void _clear() {
19     memset(Head, 0, sizeof(Head[0]) * (n + 2));
20     ne = 0;
21 }
22
23 void BFS(int src) {
24     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
25     memset(Par, -1, sizeof(Par[0]) * (n + 2));
26
27     queue <int> q;
28     q.push(src);
29     dis[src] = 0;
30
31     int u;
32     while(q.size()) {
33         u = q.front(); q.pop();
34         for(int i = Head[u]; i; i = Next[i])
35             if(dis[To[i]] == oo) {
36                 dis[To[i]] = dis[u] + 1;
```

```

37   Par[To[i]] = u;
38   q.push(To[i]);
39   }
40 }
41 }
42
43 int main()
44 {
45   cin >> n >> m >> st >> tr;
46   while(m--) {
47     cin >> u >> v >> tax;
48     addEdge(u, v, tax);
49     addEdge(v, u, tax);
50   }
51
52   BFS(st);
53   cout << dis[tr] << endl;
54 }

```

## 2.7 Connected components

```

1  const int N = 1e5 + 9, M = 1e6 + 9;
2
3  int Head[N], Next[M], To[M], ne, u, v, n, m, CCs;
4  bool visited[N];
5
6  void addEdge(int from, int to) {
7    Next[++ne] = Head[from];
8    Head[from] = ne;
9    To[ne] = to;
10 }
11
12 void DFS(int node) {
13   visited[node] = true;
14   for(int e = Head[node]; e; e = Next[e])
15     if(!visited[To[e]])
16       DFS(To[e]);
17 }
18
19 int main() {
20   cin >> n >> m;
21   while(m--) {
22     cin >> u >> v;
23     addEdge(u, v);
24     addEdge(v, u);
25   }
26
27   for(int node = 1; node <= n; ++node) if(!visited[node])
28     ++CCs, DFS(node);
29
30   cout << CCs << endl;
31 }

```

## 2.8 Cycle detection (directed graph)

```

1  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
5  ll dis[N];
6  bool hasCycle;
7  char visited[N];
8
9  void addEdge(int from, int to) {
10   Next[++ne] = Head[from];
11   Head[from] = ne;
12   To[ne] = to;
13 }
14
15 void DFS(int node) {
16   if(hasCycle != visited[node] == 1)
17     return; /* Oops, revisiting active node */
18   visited[node] = 1; /* current node legend mode has been activated */
19
20   for(int i = Head[node]; i; i = Next[i])
21     if(visited[To[i]] != 2)
22       DFS(To[i]);
23
24   visited[node] = 2; /* done with this node and mark it as visited */
25 }
26
27 int main() {

```

```

28   cin >> n >> m;
29   while(m--) {
30     cin >> u >> v;
31     addEdge(u, v);
32   }
33
34   for(int i = 1; i <= n; ++i)
35     if(!visited[i])
36       DFS(i);
37
38   cout << (hasCycle ? "YES" : "NO") << endl;
39 }

```

## 2.9 Cycle detection (undirected graph)

```

1  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
5  ll dis[N];
6  bool visited[N], hasCycle;
7
8  void addEdge(int from, int to) {
9    Next[++ne] = Head[from];
10   Head[from] = ne;
11   To[ne] = to;
12 }
13
14 void DFS(int node, int parent = -1) {
15   if(hasCycle != visited[node])
16     return;
17   visited[node] = true;
18
19   for(int i = Head[node]; i; i = Next[i])
20     if(To[i] != parent)
21       DFS(To[i], node);
22 }
23
24 int main() {
25   cin >> n >> m;
26   while(m--) {
27     cin >> u >> v;
28     addEdge(u, v);
29     addEdge(v, u);
30   }
31
32   for(int i = 1; i <= n; ++i)
33     if(!visited[i])
34       DFS(i);
35
36   cout << (hasCycle ? "YES" : "NO") << endl;
37 }

```

## 2.10 Depth first search (dfs)

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  typedef int64_t ll;
4
5  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
6  ll INF = 0x3f3f3f3f3f3f3f3f;
7
8  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
9  ll dis[N];
10 bool vis[N];
11
12 void addEdge(int from, int to, int cost) {
13   Next[++ne] = Head[from];
14   Head[from] = ne;
15   Cost[ne] = cost;
16   To[ne] = to;
17 }
18
19 void _clear() {
20   memset(Head, 0, sizeof(Head[0]) * (n + 2));
21   ne = 0;
22 }
23
24 void DFS(int node) {
25   vis[node] = true;
26   for(int i = Head[node]; i; i = Next[i])
27     if(!vis[To[i]])

```

```

28     DFS(To[i]);
29 }
30
31 int main() {
32     cin >> n >> m;
33     while(m--) {
34         cin >> u >> v;
35         addEdge(u, v);
36         addEdge(v, u);
37     }
38
39     for(int i = 1; i <= n; ++i)
40         if(!vis[i])
41             DFS(i);
42 }

```

## 2.11 Dijkstra (dense graph)

```

1  /** Dijkstra on dense graphs
2   * complexity :  $O(n^2 + m)$ 
3   */
4
5  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
6  ll INF = 0x3f3f3f3f3f3f3f3f;
7
8  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
9  ll dis[N];
10
11 void addEdge(int from, int to, int cost) {
12     Next[++ne] = Head[from];
13     Head[from] = ne;
14     Cost[ne] = cost;
15     To[ne] = to;
16 }
17
18 void _clear() {
19     memset(Head, 0, sizeof(Head[0]) * (n + 2));
20     ne = 0;
21 }
22
23 void Dijkstra(int src, int V) {
24     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
25     memset(Par, -1, sizeof(Par[0]) * (n + 2));
26
27     vector<bool> mark(V + 1, false);
28
29     dis[src] = 0;
30     for(int i = 1; i <= V; ++i) {
31         int node = 0;
32         for(int j = 1; j <= V; ++j)
33             if(!mark[j] && dis[j] < dis[node])
34                 node = j;
35
36         if(dis[node] == INF) break;
37
38         mark[node] = true;
39         for(int i = Head[node]; i; i = Next[i])
40             if(dis[node] + Cost[i] < dis[To[i]]) {
41                 dis[To[i]] = dis[node] + Cost[i];
42                 Par[To[i]] = node;
43             }
44     }
45 }

```

## 2.12 Dijkstra (grid)

```

1  const int dr[] = { 1, -1, 0, 0, 1, 1, -1, -1 };
2  const int dc[] = { 0, 0, 1, -1, 1, -1, 1, -1 };
3  const char dir[] = {'D', 'U', 'R', 'L'};
4
5  const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
6
7  int grid[N][N], dis[N][N], n, m;
8
9  bool valid(int r, int c) {
10     return r >= 1 && r <= n && c >= 1 && c <= m;
11 }
12
13 void Dijkstra(int sr, int sc) {
14     memset(dis, 0x3f, sizeof(dis)); // memset(dis, 0x3f, n * m) we don't do that here
15
16     priority_queue<tuple<int, int, int>> Q;

```

```

17
18     dis[sr][sc] = grid[sr][sc];
19     Q.push({-grid[sr][sc], sr, sc});
20
21     int cost, r, c, nr, nc;
22     while(Q.size()) {
23         tie(cost, r, c) = Q.top(); Q.pop();
24         if((-cost) > dis[r][c]) continue; // lazy deletion
25
26         for(int i = 0; i < 4; ++i) {
27             nr = r + dr[i];
28             nc = c + dc[i];
29
30             if(!valid(nr, nc)) continue;
31
32             if(dis[r][c] + grid[nr][nc] < dis[nr][nc]) {
33                 dis[nr][nc] = dis[r][c] + grid[nr][nc];
34                 Q.push({-dis[nr][nc], nr, nc});
35             }
36         }
37     }
38 }

```

## 2.13 Dijkstra (negative weighted graph)

```

1  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
5  ll dis[N];
6
7  void addEdge(int from, int to, int cost) {
8     Next[++ne] = Head[from];
9     Head[from] = ne;
10     Cost[ne] = cost;
11     To[ne] = to;
12 }
13
14 void _clear() {
15     memset(Head, 0, sizeof(Head[0]) * (n + 2));
16     ne = 0;
17 }
18
19 void Dijkstra(int src) {
20     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
21     memset(Par, -1, sizeof(Par[0]) * (n + 2));
22
23     priority_queue<pair<ll, int>> Q;
24
25     dis[src] = 0;
26     Q.push({-dis[src], src});
27
28     int node;
29     ll cost;
30     while(Q.size()) {
31         tie(cost, node) = Q.top(); Q.pop();
32         if((-cost) > dis[node]) continue;
33
34         for(int i = Head[node]; i; i = Next[i])
35             if(dis[node] + Cost[i] < dis[To[i]]) {
36                 dis[To[i]] = dis[node] + Cost[i];
37                 Q.push({-dis[To[i]], To[i]});
38                 Par[To[i]] = node;
39             }
40     }
41 }

```

## 2.14 Dijkstra (sparse graph)

```

1  /** Dijkstra on sparse graphs
2   * - complexity :  $O(n + m \log n \rightarrow O(n \log n + m)$ 
3   * - Single Source Single Destination Shortest Path Problem
4   * - Positive Weight Edges only
5   * - Subpaths of shortest paths from u to v are shortest paths!
6   */
7
8  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
9  ll INF = 0x3f3f3f3f3f3f3f3f;
10
11 int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
12 ll dis[N];
13

```

```

14 void addEdge(int from, int to, int cost) {
15     Next[++ne] = Head[from];
16     Head[from] = ne;
17     Cost[ne] = cost;
18     To[ne] = to;
19 }
20
21 void _clear() {
22     memset(Head, 0, sizeof(Head[0]) * (n + 2));
23     ne = 0;
24 }
25
26 void Dijkstra(int src, int trg) {
27     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
28     memset(Par, -1, sizeof(Par[0]) * (n + 2));
29
30     priority_queue <pair <ll, int> > Q;
31
32     dis[src] = 0;
33     Q.push({-dis[src], src});
34
35     int node;
36     ll cost;
37     while(Q.size()) {
38         tie(cost, node) = Q.top(); Q.pop();
39
40         if((-cost) > dis[node]) continue; // lazy deletion
41         if(node == trg) return;         // cheapest cost in case of positive weight edges
42
43         for(int i = Head[node]; i; i = Next[i])
44             if(dis[node] + Cost[i] < dis[To[i]]) {
45                 dis[To[i]] = dis[node] + Cost[i];
46                 Q.push({-dis[To[i]], To[i]});
47                 Par[To[i]] = node;
48             }
49     }
50 }

```

```

48 }
49
50 if(dfs_num[node] == dfs_low[node]) {
51     ++ID;
52     for(int cur = -1; cur ^ node; cur = Next[cur]) {
53         in_stack[cur = Stack[--top]] = false;
54         compID[cur] = ID;
55         ++compSize[ID];
56     }
57 }
58
59 void Tarjan() {
60     for(int i = 1; i <= n; ++i)
61         if(dfs_num[i] == 0)
62             Tarjan(i);
63 }
64
65 void DFS(int node) {
66     dfs_num[node] = 1;
67     for(int i = Head[node]; i; i = Next[i]) {
68         if(compID[node] != compID[To[i]])
69             addEdgeDAG(compID[node], compID[To[i]]);
70
71         if(dfs_num[To[i]] == 0)
72             DFS(To[i]);
73     }
74 }
75
76 void construct_dag() {
77     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
78
79     for(int i = 1; i <= n; ++i)
80         if(dfs_num[i] == 0)
81             DFS(i);
82 }
83

```

## 2.15 Directed cyclic graph into acyclic

```

1  const int N = 1e5 + 9, M = 2e6 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], To[M], Next[M], Cost[M];
5  int dfs_num[N], dfs_low[N], out[N];
6  int Stack[N], compID[N], compSize[N];
7  int ne, n, m, u, v, w;
8  int dfs_timer, top, ID;
9  bool in_stack[N];
10
11 int HeadDAG[N], ToDAG[M], NextDAG[M], CostDAG[M], neDAG;
12
13 void addEdge(int from, int to, int cost = 0) {
14     Next[++ne] = Head[from];
15     Head[from] = ne;
16     Cost[ne] = cost;
17     To[ne] = to;
18 }
19
20 void addEdgeDAG(int from, int to, int cost = 0) {
21     NextDAG[++neDAG] = HeadDAG[from];
22     HeadDAG[from] = neDAG;
23     CostDAG[ne] = cost;
24     ToDAG[neDAG] = to;
25     ++out[from];
26 }
27
28 void _clear() {
29     memset(Head, 0, sizeof(Head[0]) * (n + 2));
30     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
31     memset(compID, 0, sizeof(compID[0]) * (n + 2));
32     memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
33     memset(HeadDAG, 0, sizeof(HeadDAG[0]) * (n + 2));
34     memset(out, 0, sizeof(out[0]) * (n + 2));
35     ne = dfs_timer = top = neDAG = ID = 0;
36 }
37
38 void Tarjan(int node) {
39     dfs_num[node] = dfs_low[node] = ++dfs_timer;
40     in_stack[Stack[top++] = node] = true;
41
42     for(int i = Head[node]; i; i = Next[i]) {
43         if(dfs_num[To[i]] == 0)
44             Tarjan(To[i]);
45
46         if(in_stack[To[i]])
47             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);

```

## 2.16 Edge classification

```

1  #pragma GCC optimize ("Ofast")
2
3  #include <bits/stdc++.h>
4
5  #define UNVISITED 0
6  #define EXPLORED 1
7  #define VISITED 2
8
9  using namespace std;
10
11 typedef int64_t ll;
12
13 const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
14 ll INF = 0x3f3f3f3f3f3f3f3f;
15
16 int Head[N], Next[M], To[M], Par[N], in_time[N], ne, n, m, u, v, dfs_timer;
17 char dfs_num[N];
18
19 void addEdge(int from, int to) {
20     Next[++ne] = Head[from];
21     Head[from] = ne;
22     To[ne] = to;
23 }
24
25 void edgeClassification(int node) {
26     dfs_num[node] = EXPLORED;
27     in_time[node] = ++dfs_timer;
28
29     for(int i = Head[node]; i; i = Next[i]) {
30         if(dfs_num[To[i]] == UNVISITED) {
31             cout << "Tree Edge : " << node << " -> " << To[i] << endl;
32
33             Par[To[i]] = node;
34             edgeClassification(To[i]);
35         }
36         else if(dfs_num[To[i]] == VISITED) {
37             /* Cross Edges only occur in directed graph */
38             if(in_time[To[i]] < in_time[node])
39                 cout << "Cross Edge : " << node << " -> " << To[i] << endl;
40             else
41                 cout << "Forward Edge : " << node << " -> " << To[i] << endl;
42         }
43         else if(dfs_num[To[i]] == EXPLORED) {
44             if(Par[node] == To[i])
45                 cout << "Bi-Directional Edge : " << node << " -> " << To[i] << endl;
46             else
47                 cout << "Backward Edge : " << node << " -> " << To[i] << " (Cycle)" << endl;
48         }

```

```

49     }
50     dfs_num[node] = VISITED;
51 }
52 }
53
54 int main() {
55     cin >> n >> m;
56     while(m--) {
57         cin >> u >> v;
58         addEdge(u, v);
59     }
60
61     for(int i = 1; i <= n; ++i)
62         if(!dfs_num[i])
63             edgeClassification(i);
64 }

```

## 2.17 Eulerian tour tree

```

1  int Head[N], To[M], Next[M], Cost[M];
2  int ne, n, m, u, v, w;
3
4  int Last[N], First[N], euler_tour[1 + N << 1];
5  ll Height[1 + N << 1];
6  int euler_timer;
7
8  void addEdge(int from, int to, int cost = 0) {
9      Next[++ne] = Head[from];
10     Head[from] = ne;
11     Cost[ne] = cost;
12     To[ne] = to;
13 }
14
15 void _clear() {
16     memset(Head, 0, sizeof(Head[0])) * (n + 2));
17     memset>Last, 0, sizeof>Last[0]) * (n + 2));
18     memset>First, 0, sizeof>First[0]) * (n + 2));
19     ne = euler_timer = 0;
20 }
21
22 /**
23  euler_tour[1 .. n * 2 - 1] = which records the sequence of visited nodes
24  Height[1 .. n * 2 - 1] = which records the depth of each visited node
25
26  First[1 .. n] = records the index of the first occurrence of node i in euler_tour
27  Last[1 .. n] = records the index of the last occurrence of node i in euler_tour
28 */
29
30 void EulerianTour(int node, ll depth = 0) {
31     euler_tour[++euler_timer] = node;
32     Height[euler_timer] = depth;
33     First[node] = euler_timer;
34
35     for(int i = Head[node]; i; i = Next[i])
36         if(First>To[i]) == 0) {
37             EulerianTour>To[i], depth + Cost[i]);
38
39             euler_tour[++euler_timer] = node;
40             Height[euler_timer] = depth;
41         }
42     Last[node] = euler_timer;
43 }
44
45 void show() {
46     for(int i = 1; i < (n << 1); ++i) cout << euler_tour[i] << " "; cout << endl;
47     for(int i = 1; i < (n << 1); ++i) cout << Height[i] << " "; cout << endl;
48     for(int i = 1; i <= n; ++i) cout << First[i] << " "; cout << endl;
49     for(int i = 1; i <= n; ++i) cout << Last[i] << " "; cout << endl;
50 }
51
52 int main() {
53     cin >> n >> m;
54     _clear();
55
56     while(m--) {
57         cin >> u >> v >> w;
58         addEdge(u, v, w);
59         addEdge(v, u, w);
60     }
61
62     EulerianTour(1);
63     show();
64 }

```

## 2.18 Flood fill

```

1  /** check if there is a path from (0, 0) to (n - 1, m - 1) using '.' only **/
2
3  int dr[4] = {1, -1, 0, 0};
4  int dc[4] = {0, 0, 1, -1};
5  char grid[N][M];
6  int n, m;
7
8  bool valid(int r, int c) {
9      return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] == '.';
10 }
11
12 bool isDis(int r, int c) {
13     return r == n - 1 && c == m - 1;
14 }
15
16 bool FloodFill(int r, int c) {
17     if(!valid(r, c)) return false;
18     if(isDis(r, c)) return true;
19
20     grid[r][c] = '#';
21     for(int i = 0; i < 4; ++i)
22         if(FloodFill(r + dr[i], c + dc[i]))
23             return true;
24     return false;
25 }
26
27 int main() {
28     cin >> n >> m;
29     for(int i = 0; i < n; ++i)
30         for(int j = 0; j < m; ++j)
31             cin >> grid[i][j];
32
33     cout << (FloodFill(0, 0) ? "YES" : "NO") << endl;
34 }

```

## 2.19 Floyd warshall (all pairs shortest path)

```

1  /** -The graph has a 'negative cycle' if at the end of the algorithm,
2      the distance from a vertex v to itself is negative.
3
4      - before k-th phase the value of d[i][j] is equal to the length of
5      the shortest path from vertex i to the vertex j,
6      if this path is allowed to enter only the vertex with numbers smaller than k
7      (the beginning and end of the path are not restricted by this property).
8  */
9
10 const int N = 500 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
11 const int64 INF = 0x3f3f3f3f3f3f3f3f;
12
13 int Par[N][N], n, m, u, v, tax;
14 int64 adj[N][N], dis[N][N];
15
16 vector<int> restorePath(int st, int tr) {
17     vector<int> path;
18     if(dis[st][tr] == INF) return path;
19
20     for(int i = tr; st ^ i; i = Par[st][i])
21         path.push_back(i);
22
23     path.push_back(st);
24     reverse(path.begin(), path.end());
25     return path;
26 }
27
28 void Floyd_Warshall() {
29     for(int i = 1; i <= n; ++i)
30         for(int j = 1; j <= n; ++j)
31             Par[i][j] = i;
32
33     for(int k = 1; k <= n; ++k)
34         for(int i = 1; i <= n; ++i)
35             for(int j = 1; j <= n; ++j)
36                 if(dis[i][k] + dis[k][j] < dis[i][j]) {
37                     dis[i][j] = dis[i][k] + dis[k][j];
38                     Par[i][j] = Par[k][j];
39                 }
40 }

```

## 2.20 Minimum spanning tree (kruskal)

```

1 class UnionFind {
2     vector<int> par;
3     vector<int> siz;
4     int num_sets;
5     size_t sz;
6
7 public:
8     UnionFind() : par(1, -1), siz(1, 1), num_sets(0), sz(0) {}
9     UnionFind(int n) : par(n + 1, -1), siz(n + 1, 1), num_sets(n), sz(n) {}
10
11     int find_set(int u) {
12         assert(u <= sz);
13
14         int leader;
15         for(leader = u; !par[leader]; leader = par[leader]);
16
17         for(int next = par[u]; u != leader; next = par[next]) {
18             par[u] = leader;
19             u = next;
20         }
21         return leader;
22     }
23
24     bool same_set(int u, int v) {
25         return find_set(u) == find_set(v);
26     }
27
28     bool union_set(int u, int v) {
29         if(same_set(u, v)) return false;
30
31         int x = find_set(u);
32         int y = find_set(v);
33
34         if(siz[x] < siz[y]) swap(x, y);
35
36         par[y] = x;
37         siz[x] += siz[y];
38
39         --num_sets;
40         return true;
41     }
42
43     int number_of_sets() {
44         return num_sets;
45     }
46
47     int size_of_set(int u) {
48         return siz[find_set(u)];
49     }
50
51     size_t size() {
52         return sz;
53     }
54
55     void clear() {
56         par.clear();
57         siz.clear();
58         sz = num_sets = 0;
59     }
60
61     void assign(size_t n) {
62         par.assign(n + 1, -1);
63         siz.assign(n + 1, 1);
64         sz = num_sets = n;
65     }
66
67     map<int, vector<int>> groups(int st) {
68         map<int, vector<int>> ret;
69
70         for(size_t i = st; i < sz + st; ++i)
71             ret[find_set(i)].push_back(i);
72
73         return ret;
74     }
75 };
76
77 int n, m, u, v, w;
78 vector< tuple<int, int, int> > edges;
79 UnionFind uf;
80
81 pair< ll, vector< pair<int, int> > > Kruskal() {
82     sort(edges.begin(), edges.end());
83
84     vector< pair<int, int> > mstEdges;
85     int from, to, cost;
86     ll minWiegth = 0;
87
88     for(tuple<int, int, int> edge : edges) {
89         tie(cost, from, to) = edge;
90         if(uf.union_set(from, to)) {
91             minWiegth += cost;
92             mstEdges.push_back(make_pair(from, to));

```

```

93     }
94 }
95
96 if(mstEdges.size() == n - 1)
97     return make_pair(minWiegth, mstEdges);
98
99 return make_pair(-1, vector< pair<int, int> > ());
100 }

```

## 2.21 Kth ancestor and lowest common ancestor (binary lift- ing)

```

1 const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2 const int LOG = 20;
3
4 int Head[N], To[M], Next[M], Par[N];
5 int up[N][LOG + 1];
6 int Log[N], Level[N];
7 int ne, n, u, v, q;
8
9 void addEdge(int from, int to) {
10     Next[++ne] = Head[from];
11     Head[from] = ne;
12     To[ne] = to;
13 }
14
15 void _clear() {
16     memset(Head, 0, sizeof(Head[0]) * (n + 2));
17     memset(Par, 0, sizeof(Par[0]) * (n + 2));
18     memset(Level, 0, sizeof(Level[0]) * (n + 2));
19     ne = 0;
20 }
21
22 int lastBit(int a) {
23     return (a & -a);
24 }
25
26 void logCalc() {
27     Log[1] = 0;
28     for(int i = 2; i < N; ++i)
29         Log[i] = Log[i >> 1] + 1;
30 }
31
32 void DFS(int node, int depth = 0) {
33     Level[node] = depth;
34     up[node][0] = Par[node]; // Par[root] = root
35
36     for(int i = 1; i <= LOG; ++i) {
37         up[node][i] = up[up[node][i - 1]][i - 1];
38     }
39
40     for(int i = Head[node]; i; i = Next[i])
41         if(To[i] != Par[node]) {
42             Par[To[i]] = node;
43             DFS(To[i], depth + 1);
44         }
45 }
46
47 int KthAncestor(int u, int k) {
48     if(k > Level[u]) return -1;
49
50     for(int i = lastBit(k); k; k -= lastBit(k), i = lastBit(k))
51         u = up[u][Log[i]];
52
53     return u;
54 }
55
56 int LCA(int u, int v) {
57     if(Level[u] < Level[v]) swap(u, v);
58     int k = Level[u] - Level[v];
59
60     u = KthAncestor(u, k);
61     if(u == v) return u;
62
63     for(int i = LOG; i >= 0; --i)
64         if(up[u][i] ^ up[v][i])
65             u = up[u][i];
66
67     u = up[u][0];
68     v = up[v][0];
69
70     return up[u][0];
71 }
72
73 int main() {
74     cin >> n;

```



```

75 _clear();
76
77 for(int i = 1; i < n; ++i) {
78     cin >> u >> v;
79     addEdge(u, v);
80     addEdge(v, u);
81 }
82
83 logCalc();
84 for(int i = 1; i <= n; ++i) if(Par[i] == 0) {
85     Par[i] = i;
86     DFS(i);
87 }
88
89 cin >> q;
90 while(q--) {
91     cin >> u >> v;
92     cout << LCA(u, v) << endl;
93 }
94 }

```

## 2.22 Lowest common ancestor (euler tour)

```

1  template <class T, class F = function <T(const T&, const T&)>>
2  class SparseTable {
3      int _N;
4      int _LOG;
5      vector <T> _A;
6      vector < vector <T> > ST;
7      vector <int> Log;
8      F func;
9
10 public :
11     SparseTable() = default;
12
13     template <class iter>
14     SparseTable(iter _begin, iter _end, const F _func = less <T>()) : func(_func) {
15         _N = distance(_begin, _end);
16         Log.assign(_N + 1, 0);
17         for(int i = 2; i <= _N; ++i)
18             Log[i] = Log[i >> 1] + 1;
19
20         _LOG = Log[_N];
21
22         _A.assign(_N + 1, 0);
23         ST.assign(_N + 1, vector <T> (_LOG + 1, 0));
24
25         _typeof(_begin) i = _begin;
26         for(int j = 1; i != _end; ++i, ++j)
27             _A[j] = *i;
28
29         build();
30     }
31
32     void build() {
33         for(int i = 1; i <= _N; ++i)
34             ST[i][0] = i;
35
36         for(int j = 1, k, d; j <= _LOG; ++j) {
37             k = (1 << j);
38             d = (k >> 1);
39
40             for(int i = 1; i + k - 1 <= _N; ++i) {
41                 T const & x = ST[i][j - 1];
42                 T const & y = ST[i + d][j - 1];
43
44                 ST[i][j] = func(_A[x], _A[y]) ? x : y;
45             }
46         }
47
48         T query(int l, int r) {
49             int d = r - l + 1;
50             T const & x = ST[l][Log[d]];
51             T const & y = ST[l + d - (1 << Log[d])][Log[d]];
52
53             return func(_A[x], _A[y]) ? x : y;
54         }
55     };
56 };
57
58 const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
59 const ll INF = 0x3f3f3f3f3f3f3f3f;
60
61 int Head[N], To[M], Next[M], Cost[M];
62 int ne, n, m, u, v, w, q;
63
64 int Last[N], First[N], euler_tour[N << 1];

```

```

65 int Height[N << 1];
66 int euler_timer;
67
68 void addEdge(int from, int to, int cost = 1) {
69     Next[++ne] = Head[from];
70     Head[from] = ne;
71     Cost[ne] = cost;
72     To[ne] = to;
73 }
74
75 void _clear() {
76     memset(Head, 0, sizeof(Head[0]) * (n + 2));
77     memset>Last, 0, sizeof>Last[0]) * (n + 2));
78     memset(First, 0, sizeof(First[0]) * (n + 2));
79     ne = euler_timer = 0;
80 }
81
82 void EulerianTour(int node, int depth = 0) {
83     euler_tour[++euler_timer] = node;
84     Height[euler_timer] = depth;
85     First[node] = euler_timer;
86
87     for(int i = Head[node]; i; i = Next[i])
88         if(First[To[i]] == 0) {
89             EulerianTour(To[i], depth + Cost[i]);
90
91             euler_tour[++euler_timer] = node;
92             Height[euler_timer] = depth;
93         }
94
95     Last[node] = euler_timer;
96 }
97
98 int main() {
99     cin >> n >> m;
100     _clear();
101
102     while(m--) {
103         cin >> u >> v;
104         addEdge(u, v);
105         addEdge(v, u);
106     }
107
108     EulerianTour(1);
109
110     SparseTable <int> st(Height + 1, Height + euler_timer + 1, [&](int a, int b) { return a <= b; });
111
112     int l, r; cin >> q;
113     while(q--) {
114         cin >> l >> r;
115
116         int left = Last[l];
117         int right = Last[r];
118         if(left > right) swap(left, right);
119
120         cout << euler_tour[ st.query(left, right) ] << endl;
121     }
122 }

```

## 2.23 Minimum vertex cover

```

1  const int N = 1e5 + 9;
2
3  int Head[N], Next[N << 1], To[N << 1], ne, u, v, n, MVC;
4
5  void addEdge(int from, int to) {
6      Next[++ne] = Head[from];
7      Head[from] = ne;
8      To[ne] = to;
9  }
10
11 bool DFS(int node, int par = -1) {
12     bool black = false;
13     for(int e = Head[node]; e; e = Next[e])
14         if(To[e] != par)
15             black |= DFS(To[e], node);
16
17     MVC += black;
18     return !black;
19 }
20
21 int main() {
22     cin >> n;
23     while(--n) {
24         cin >> u >> v;
25         addEdge(u, v);
26         addEdge(v, u);

```

```

27 }
28
29 DFS(1);
30 cout << MVC << endl;
31 }

```

## 2.24 Restoring the path

```

1  const int dr []      = {-1, 0, 1, 0};
2  const int dc []      = {0, 1, 0, -1};
3  const char dir []    = {'U', 'R', 'D', 'L'};
4  map<char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};
5
6  /** Implicit Graphs
7
8      - in BFS, Dijkstra or Bellman-Ford function write -> Par[nr][nc] = dir[i ^ 2]
9      - char Par[N][N] initialize with -1
10     - si start i
11     - sj strat j
12     - fi target i
13     - fj target j
14     - char dir and its map inv
15     - dr, dc
16  */
17
18 string restorePath(int si, int sj, int fi, int fj) {
19     string s;
20     if(Par[ei][ej] == -1) return s;
21
22     int ei = fi, ej = fj;
23     for(char i = Par[fi][fj]; (si ^ fi) || (sj ^ fj); i = Par[fi][fj]) {
24         s += dir[inv[i] ^ 2];
25         fi += dr[inv[i]];
26         fj += dc[inv[i]];
27     }
28
29     reverse(s.begin(), s.end());
30     return s;
31 }
32
33 /** Explicit Graphs (BFS, Dijkstra or Bellman-Ford)
34
35     - int Par[N] initialize with -1
36     - ll dis[N] initialize with 0x3f
37     - ll INF = 0x3f3f3f3f3f3f3f3f
38  */
39
40 vector<int> restorePath(int dest) {
41     vector<int> path;
42     if(dis[dest] == INF) return path;
43
44     for(int i = dest; ~i; i = Par[i])
45         path.push_back(i);
46
47     reverse(path.begin(), path.end());
48     return path;
49 }
50
51 /** in case of Floyd-Warshall:
52
53     - ll dis[N][N] initialize with 0x3f
54     - ll INF = 0x3f3f3f3f3f3f3f3f
55     - int Par[N][N] initialize with      Par[i][j] = i;
56     - in Floyd-Warshall function write -> Par[i][j] = Par[k][j];
57  */
58
59 vector<int> restorePath(int st, int tr) {
60     vector<int> path;
61     if(dis[st][tr] == INF) return path;
62
63     for(int i = tr; st ^ i; i = Par[st][i])
64         path.push_back(i);
65
66     path.push_back(st);
67     reverse(path.begin(), path.end());
68     return path;
69 }

```

## 2.25 Shortest path faster algorithm1 (spfa)

```

1  /** Shortest Path Faster Algorithm :
2      - This algorithm runs in O(kE) where k is a number depending on the graph.

```

```

3      - The maximum k can be V (which is the same as the time complexity of Bellman Fords).
4      - However, in practice SPFA (which uses a queue) is as fast as Dijkstras (which uses a priority
5        queue).
6      - SPFA can deal with negative weight edge. If the graph has no negative cycle, SPFA runs well on
7        it.
8      - If the graph has negative cycle(s), SPFA can also detect it as there must be some vertex (those
9        on the negative cycle)
10     that enters the queue for over V , 1 times.
11
12  */
13
14 const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
15 ll INF = 0x3f3f3f3f3f3f3f3f;
16 ll mINF = 0xc0c0c0c0c0c0c0c0;
17
18 int Head[N], Par[N], Next[M], To[M], Cost[M], Cnt[N], ne, n, m, u, v, st, tax;
19 ll dis[N];
20 bool Inq[N];
21
22 void addEdge(int from, int to, int cost) {
23     Next[++ne] = Head[from];
24     Head[from] = ne;
25     Cost[ne] = cost;
26     To[ne] = to;
27 }
28
29 void _clear() {
30     memset(Head, 0, sizeof(Head[0]) * (n + 2));
31     ne = 0;
32 }
33
34 void _set() {
35     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
36     memset(Par, -1, sizeof(Par[0]) * (n + 2));
37     memset(Cnt, 0, sizeof(Cnt[0]) * (n + 2));
38     memset(Inq, 0, sizeof(Inq[0]) * (n + 2));
39 }
40
41 bool SPFA(int src) {
42     _set();
43
44     deque<int> Q;
45     Q.push_front(src);
46
47     dis[src] = 0;
48     Cnt[src] = 1;
49     Inq[src] = 1;
50
51     int node;
52     while(Q.size()) {
53         node = Q.front(); Q.pop_front(); Inq[node] = 0;
54
55         for(int i = Head[node]; i; i = Next[i])
56             if(dis[node] + Cost[i] < dis[To[i]]) {
57                 dis[To[i]] = dis[node] + Cost[i];
58                 Par[To[i]] = node;
59
60                 if(!Inq[To[i]]) {
61                     if(++Cnt[To[i]] == n)
62                         return true; // graph has a negative weight cycle
63
64                     if(Q.size() && dis[To[i]] > dis[Q.front()])
65                         Q.push_back(To[i]);
66                     else
67                         Q.push_front(To[i]);
68
69                     Inq[To[i]] = true;
70                 }
71             }
72     }
73     return false;
74 }

```

## 2.26 Single source shortest path

```

1  const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], Par[N], Next[M], To[M], ne, n, m, u, v, st, tr;
5  ll dis[N];
6
7  void addEdge(int from, int to) {
8      Next[++ne] = Head[from];
9      Head[from] = ne;
10     To[ne] = to;
11 }
12
13 void _clear() {

```

```

14  memset(Head, 0, sizeof(Head[0]) * (n + 2));
15  ne = 0;
16  }
17
18  void BFS(int src) {
19      memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
20      memset(Par, -1, sizeof(Par[0]) * (n + 2));
21
22      queue<int> Q;
23      Q.push(src);
24      dis[src] = 0;
25
26      int node;
27      while(Q.size()) {
28          node = Q.front(); Q.pop();
29          for(int i = Head[node]; i; i = Next[i]) if(dis[To[i]] == INF) {
30              dis[To[i]] = dis[node] + 1;
31              Par[To[i]] = node;
32              Q.push(To[i]);
33          }
34      }
35  }

```

## 2.27 Single source shortest path (grid)

```

1  const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  const int dr []   = {-1, 0, 1, 0};
5  const int dc []   = {0, 1, 0, -1};
6  const char dir [] = {'U', 'R', 'D', 'L'};
7  map<char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};
8
9  int dis[N][N], n, m;
10 char Par[N][N];
11
12 bool valid(int r, int c) {
13     return r >= 1 && r <= n && c >= 1 && c <= m && dis[r][c] == oo;
14 }
15
16 void BFS(int sr, int sc) {
17     memset(dis, 0x3f, sizeof(dis));
18     memset(Par, -1, sizeof(Par));
19
20     queue< pair<int, int> > Q;
21     dis[sr][sc] = 0;
22     Q.push({sr, sc});
23
24     int r, c, nr, nc;
25     while(Q.size()) {
26         tie(r, c) = Q.front(); Q.pop();
27
28         for(int i = 0; i < 4; ++i) {
29             nr = r + dr[i];
30             nc = c + dc[i];
31
32             if(!valid(nr, nc)) continue;
33
34             dis[nr][nc] = dis[r][c] + 1;
35             Par[nr][nc] = dir[i ^ 2];
36             Q.push({nr, nc});
37         }
38     }
39 }

```

## 2.28 Tarjan (strongly connected components)

```

1  const int N = 1e5 + 9, M = 2e6 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2  ll INF = 0x3f3f3f3f3f3f3f3f;
3
4  int Head[N], To[M], Next[M], Cost[M];
5  int dfs_num[N], dfs_low[N];
6  int Stack[N], compID[N], compSize[N];
7  int ne, n, m, u, v, w;
8  int dfs_timer, top, ID;
9  bool in_stack[N];
10
11 void addEdge(int from, int to, int cost = 0) {
12     Next[++ne] = Head[from];
13     Head[from] = ne;
14     Cost[ne] = cost;
15     To[ne] = to;

```

```

16 }
17
18 void _clear() {
19     memset(Head, 0, sizeof(Head[0]) * (n + 2));
20     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
21     memset(compID, 0, sizeof(compID[0]) * (n + 2));
22     memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
23     ne = dfs_timer = top = ID = 0;
24 }
25
26 void Tarjan(int node) {
27     dfs_num[node] = dfs_low[node] = ++dfs_timer;
28     in_stack[Stack[top++] = node] = true;
29
30     for(int i = Head[node]; i; i = Next[i]) {
31         if(dfs_num[To[i]] == 0)
32             Tarjan(To[i]);
33
34         if(in_stack[To[i]])
35             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
36     }
37
38     if(dfs_num[node] == dfs_low[node]) {
39         ++ID;
40         for(int cur = -1; cur ^ node; cur = Stack[--top]) {
41             in_stack[cur] = false;
42             compID[cur] = ID;
43             ++compSize[ID];
44         }
45     }
46 }
47
48 void Tarjan() {
49     for(int i = 1; i <= n; ++i)
50         if(dfs_num[i] == 0)
51             Tarjan(i);
52 }

```

## 2.29 Topological sort (dfs)

```

1  int Head[N], Next[M], To[M], ne, n, m, u, v;
2  bool vis[N];
3  vector<int> t_sort;
4
5  void addEdge(int from, int to) {
6     Next[++ne] = Head[from];
7     Head[from] = ne;
8     To[ne] = to;
9 }
10
11 void DFS(int node) {
12     vis[node] = true;
13     for(int i = Head[node]; i; i = Next[i])
14         if(!vis[To[i]])
15             DFS(To[i]);
16
17     t_sort.push_back(node);
18 }
19
20 vector<int> topological_sort(int n) {
21
22     t_sort.clear();
23     for(int i = 1; i <= n; ++i) if(!vis[i])
24         DFS(i);
25
26     reverse(t_sort.begin(), t_sort.end());
27     return t_sort;
28 }
29
30 int main() {
31     cin >> n >> m;
32     while(m--) {
33         cin >> u >> v;
34         addEdge(u, v);
35     }
36
37     vector<int> v = topological_sort(n);
38     for(int i : v)
39         cout << i << ' ';
40 }

```

## 2.30 Topological sort (kahns algorithm)

```

1 int Head[N], Next[M], To[M], in[N], ne, n, m, u, v;
2
3 void addEdge(int from, int to) {
4     Next[++ne] = Head[from];
5     Head[from] = ne;
6     To[ne] = to;
7 }
8
9 vector<int> kahn(int n) {
10     vector<int> ready, ret;
11
12     for(int i = 1; i <= n; ++i)
13         if(!in[i])
14             ready.push_back(i);
15
16     int node;
17     while(!ready.empty()) {
18         node = ready.back(); ready.pop_back();
19         ret.push_back(node);
20
21         for(int i = Head[node]; i; i = Next[i])
22             if(--in[To[i]] == 0)
23                 ready.push_back(To[i]);
24     }
25     return ret;
26 }
27
28 int main() {
29     cin >> n >> m;
30     while(m--) {
31         cin >> u >> v;
32         addEdge(u, v);
33         ++in[v];
34     }
35
36     vector<int> v = kahn(n);
37     if((int)v.size() == n) for(int i : v)
38         cout << i << ' ';
39     else
40         cout << "not a DAG!" << endl;
41 }

```

## 2.31 Tree diameter

```

1 const int N = 3e5 + 9, M = 6e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2 ll INF = 0x3f3f3f3f3f3f3f3f;
3
4 int Head[N], Next[M], To[M], Par[N], toLeaf[N], maxLength[N], ne, n, m, u, v, w;
5
6 void addEdge(int from, int to) {
7     Next[++ne] = Head[from];
8     Head[from] = ne;
9     To[ne] = to;
10 }
11
12 void _clear() {
13     memset(Head, 0, sizeof(Head[0]) * (n + 2));
14     memset(Par, -1, sizeof(Par[0]) * (n + 2));
15     ne = 0;
16 }
17
18 void dfs_toLeaf(int node, int par = -1)
19 {
20     toLeaf[node] = 0;
21     for(int i = Head[node]; i; i = Next[i])
22         if(To[i] != par) {
23             dfs_toLeaf(To[i], node);
24             if(toLeaf[To[i]] + 1 > toLeaf[node])
25                 toLeaf[node] = toLeaf[To[i]] + 1;
26         }
27 }
28
29 void dfs_maxLength(int node, int par = -1)
30 {
31     int firstMax = -1;
32     int secondMax = -1;
33     for(int i = Head[node]; i; i = Next[i])
34         if(To[i] != par) {
35             dfs_maxLength(To[i], node);
36
37             if(toLeaf[To[i]] > firstMax) {
38                 if(firstMax > secondMax)
39                     secondMax = firstMax;
40                 firstMax = toLeaf[To[i]];
41             } else if(toLeaf[To[i]] > secondMax)
42                 secondMax = toLeaf[To[i]];
43 }

```

```

44     maxLength[node] = firstMax + secondMax + 2;
45 }
46
47 void Solve()
48 {
49     cin >> n;
50     _clear();
51
52     for(int i = 1; i < n; ++i) {
53         cin >> u >> v;
54         addEdge(u, v);
55         addEdge(v, u);
56     }
57
58     dfs_toLeaf(1);
59     dfs_maxLength(1);
60
61     int diameter = 0;
62     for(int i = 1; i <= n; ++i)
63         if(maxLength[i] > diameter)
64             diameter = maxLength[i];
65
66     cout << diameter << endl;
67 }

```

## 2.32 0-1 bfs

```

1 const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
2 ll INF = 0x3f3f3f3f3f3f3f3f;
3
4 int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
5 ll dis[N];
6
7 void addEdge(int from, int to, int cost) {
8     Next[++ne] = Head[from];
9     Head[from] = ne;
10     Cost[ne] = cost;
11     To[ne] = to;
12 }
13
14 void _clear() {
15     memset(Head, 0, sizeof(Head[0]) * (n + 2));
16     ne = 0;
17 }
18
19 void BFS(int src, int trg) {
20     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
21     memset(Par, -1, sizeof(Par[0]) * (n + 2));
22
23     deque<int> Q;
24     Q.push_front(src);
25     dis[src] = 0;
26
27     int node;
28     while(Q.size()) {
29         node = Q.front(); Q.pop_front();
30         if(node == trg) return;
31
32         for(int i = Head[node]; i; i = Next[i])
33             if(dis[node] + Cost[i] < dis[To[i]]) {
34                 dis[To[i]] = dis[node] + Cost[i];
35                 if(Cost[i])
36                     Q.push_back(To[i]);
37                 else
38                     Q.push_front(To[i]);
39             }
40     }
41 }

```

## 2.33 0-1 bfs (grid)

```

1 const int dr[] = { -1, -1, 0, 1, 1, 1, 0, -1 };
2 const int dc[] = { 0, 1, 1, 1, 0, -1, -1, -1 };
3 const char dir[] = {'D', 'U', 'R', 'L'};
4
5 const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
6
7 int dis[N][N], n, m, si, sj, ti, tj;
8 char grid[N][N];
9
10 bool valid(int r, int c) {
11     return r >= 1 && r <= n && c >= 1 && c <= m;

```

```

12 }
13
14 int ZBFS(int sr, int sc, int tr, int tc) {
15     memset(dis, 0x3f, sizeof(dis)); // memset(dis, 0x3f, n * m) we don't do that here
16
17     deque <pair <int, int> > Q;
18
19     dis[sr][sc] = 0;
20     Q.push_front({sr, sc});
21
22     int r, c, nr, nc, ncost;
23     while(Q.size()) {
24         tie(r, c) = Q.front(); Q.pop_front();
25         if(r == tr && c == tc) return dis[r][c];
26
27         for(int i = 0; i < 8; ++i) {
28             nr = r + dr[i];
29             nc = c + dc[i];
30
31             if(!valid(nr, nc)) continue;
32             ncost = (i != grid[r][c]);
33
34             if(dis[r][c] + ncost < dis[nr][nc]) {
35                 dis[nr][nc] = dis[r][c] + ncost;
36
37                 if(ncost)
38                     Q.push_back({nr, nc});
39                 else
40                     Q.push_front({nr, nc});
41             }
42         }
43     }
44     return oo;
45 }

```

## 3 Data structures

### 3.1 Union find disjoint sets

```

1 /**
2  * Maintain a set of elements partitioned into non-overlapping subsets. Each
3  * partition is assigned a unique representative known as the parent, or root. The
4  * following implements two well-known optimizations known as union-by-size and
5  * path compression. This version is simplified to only work on integer elements.
6  *
7  * - find_set(u) returns the unique representative of the partition containing u.
8  * - same_set(u, v) returns whether elements u and v belong to the same partition.
9  * - union_set(u, v) replaces the partitions containing u and v with a single new
10   * partition consisting of the union of elements in the original partitions.
11  *
12  * Time Complexity:
13  * - O(a(n)) per call to find_set(), same_set(), and union_set(), where n is the
14  * number of elements, and a(n) is the extremely slow growing inverse of the Ackermann function
15  * (effectively a very small constant for all practical values of n).
16  *
17  * Space Complexity:
18  * - O(n) for storage of the disjoint set forest elements.
19  * - O(1) auxiliary for all operations.
20  */
21
22 class UnionFind {
23     vector<int> par;
24     vector<int> siz;
25     int num_sets;
26     size_t sz;
27
28 public:
29     UnionFind() : par(1, -1), siz(1, 1), num_sets(0), sz(0) {}
30     UnionFind(int n) : par(n + 1, -1), siz(n + 1, 1), num_sets(n), sz(n) {}
31
32     int find_set(int u) {
33         assert(u <= sz);
34
35         int leader;
36         for(leader = u; !par[leader]; leader = par[leader]);
37
38         for(int next = par[u]; u != leader; next = par[next]) {
39             par[u] = leader;
40             u = next;
41         }
42         return leader;
43     }
44
45     bool same_set(int u, int v) {

```

```

46         return find_set(u) == find_set(v);
47     }
48
49     bool union_set(int u, int v) {
50         if(same_set(u, v)) return false;
51
52         int x = find_set(u);
53         int y = find_set(v);
54
55         if(siz[x] < siz[y]) swap(x, y);
56
57         par[y] = x;
58         siz[x] += siz[y];
59
60         --num_sets;
61         return true;
62     }
63
64     int number_of_sets() {
65         return num_sets;
66     }
67
68     int size_of_set(int u) {
69         return siz[find_set(u)];
70     }
71
72     size_t size() {
73         return sz;
74     }
75
76     void clear() {
77         par.clear();
78         siz.clear();
79         sz = num_sets = 0;
80     }
81
82     void assign(size_t n) {
83         par.assign(n + 1, -1);
84         siz.assign(n + 1, 1);
85         sz = num_sets = n;
86     }
87
88     map<int, vector<int>> groups(int st) {
89         map<int, vector<int>> ret;
90
91         for(size_t i = st; i < sz + st; ++i)
92             ret[find_set(i)].push_back(i);
93
94         return ret;
95     }
96 };

```

### 3.2 Segment tree (rmq)

```

1 /** resources:
2  * 1. https://codeforces.com/blog/entry/15729
3  * 2. https://codeforces.com/blog/entry/15890
4  * 3. https://codeforces.com/blog/entry/18051
5  * 4. https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/practice-problems/algorithm/range-minimum-query/description/
6  */
7
8 template <class T, class F = function <T(const T &, const T &)> >
9 class SegmentTree {
10     vector<T> A;
11     vector<T> ST;
12     vector<T> LT;
13     F func;
14     int _N;
15
16 public:
17     template <class iter>
18     SegmentTree(iter _begin, iter _end, const F _func = [](T a, T b) {return a <= b ? a : b;}) : func(
19         _func) {
20         _N = distance(_begin, _end);
21         _N = (1 << ((int) ceil(log2(_N))));
22
23         A.assign(_N + 1, 0);
24         ST.assign(_N << 1, 0);
25         LT.assign(_N << 1, 0);
26
27         _type_of(_begin) i = _begin;
28         for(int j = 1; i != _end; ++i, ++j)
29             A[j] = *i;
30
31         build(1, 1, _N);
32     }

```

### 3.3 Merge sort tree

```

32 void build(int p, int l, int r) {
33     if(l == r) {
34         ST[p] = _A[l];
35         return;
36     }
37
38     int mid = (l + r) >> 1;
39
40     build(p + p, l, mid);
41     build(p + p + 1, mid + 1, r);
42
43     const T & x = ST[p + p];
44     const T & y = ST[p + p + 1];
45
46     ST[p] = func(x, y);
47 }
48
49 void update_range(int ul, int ur, int delta) {
50     update_range(ul, ur, delta, 1, 1, _N);
51 }
52
53 T query(int ql, int qr) {
54     return query(ql, qr, 1, 1, _N);
55 }
56
57 void update_point(int inx, int delta) {
58     inx += _N - 1;
59     ST[inx] = delta;
60
61     while(inx > 1) {
62         inx >>= 1;
63
64         const T & x = ST[inx + inx];
65         const T & y = ST[inx + inx + 1];
66
67         ST[inx] = func(x, y);
68     }
69 }
70
71 private :
72 void update_range(int ul, int ur, int delta, int p, int l, int r) {
73     if(r < ul || ur < l)
74         return;
75
76     if(ul <= l && r <= ur) {
77         ST[p] += delta;
78         LT[p] += delta;
79         return;
80     }
81
82     propagate(p);
83
84     int mid = (l + r) >> 1;
85
86     update_range(ul, ur, delta, p + p, l, mid);
87     update_range(ul, ur, delta, p + p + 1, mid + 1, r);
88
89     const T & x = ST[p + p];
90     const T & y = ST[p + p + 1];
91
92     ST[p] = func(x, y);
93 }
94
95 T query(int ql, int qr, int p, int l, int r) {
96     if(r < ql || qr < l)
97         return INT_MAX;
98
99     if(ql <= l && r <= qr)
100         return ST[p];
101
102     propagate(p);
103
104     int mid = (l + r) >> 1;
105
106     const T & x = query(ql, qr, p + p, l, mid);
107     const T & y = query(ql, qr, p + p + 1, mid + 1, r);
108
109     return func(x, y);
110 }
111
112 void propagate(int p) {
113     if(LT[p]) {
114         ST[p + p] += LT[p];
115         ST[p + p + 1] += LT[p];
116         LT[p + p] += LT[p];
117         LT[p + p + 1] += LT[p];
118         LT[p] = 0;
119     }
120 }
121 }
122 };

```

```

1  /** https://www.spoj.com/problems/KQUERY/
2  **/
3
4  class SegmentTree {
5      vector <vector <int>> > sTree;
6      vector <int> localArr;
7      int NP2, oo = 0x3f3f3f3f;
8
9  public :
10     template <class T>
11     SegmentTree(T _begin, T _end) {
12         NP2 = 1;
13         int n = _end - _begin;
14         while(NP2 < n) NP2 <=<= 1;
15
16         sTree.assign(NP2 << 1, vector <int> ());
17         localArr.assign(NP2 + 1, 0);
18
19         _typeof(_begin) i = _begin;
20         for(int j = 1; i != _end; i++, ++j)
21             localArr[j] = *i;
22
23         build(1, 1, NP2);
24     }
25
26     void build(int p, int l, int r) {
27         if(l == r) {
28             sTree[p].push_back(localArr[l]);
29             return;
30         }
31
32         build(left(p), l, mid(l, r));
33         build(right(p), mid(l, r) + 1, r);
34
35         merge(p);
36     }
37
38     int query(int ql, int qr, int k) {
39         return query(ql, qr, k, 1, 1, NP2);
40     }
41
42 private :
43     int query(int ql, int qr, int k, int p, int l, int r) {
44         if(isOutside(ql, qr, l, r))
45             return 0;
46
47         if(isInside(ql, qr, l, r)) {
48             return sTree[p].end() - upper_bound(sTree[p].begin(), sTree[p].end(), k);
49         }
50
51         return query(ql, qr, k, left(p), l, mid(l, r)) + query(ql, qr, k, right(p), mid(l, r) + 1, r);
52     }
53
54     void merge(int p) {
55         vector <int> & L = sTree[left(p)];
56         vector <int> & R = sTree[right(p)];
57
58         int l_size = L.size();
59         int r_size = R.size();
60         int p_size = l_size + r_size;
61
62         L.push_back(INT_MAX);
63         R.push_back(INT_MAX);
64
65         sTree[p].resize(p_size);
66
67         for(int k = 0, i = 0, j = 0; k < p_size; ++k)
68             if(L[i] <= R[j])
69                 sTree[p][k] = L[i], i += (L[i] != INT_MAX);
70             else
71                 sTree[p][k] = R[j], j += (R[j] != INT_MAX);
72
73         L.pop_back();
74         R.pop_back();
75     }
76
77     inline bool isInside(int ql, int qr, int sl, int sr) {
78         return (ql <= sl && sr <= qr);
79     }
80
81     inline bool isOutside(int ql, int qr, int sl, int sr) {
82         return (sr < ql || qr < sl);
83     }
84
85     inline int mid (int l, int r) {

```

```

86     return ((l + r) >> 1);
87 }
88
89 inline int left(int p) {
90     return (p << 1);
91 }
92
93 inline int right(int p) {
94     return ((p << 1) | 1);
95 }
96 };

```

### 3.4 Sparse table (rmq)

```

1  template <class T, class F = function <T(const T&, const T&)>>
2  class SparseTable {
3      int _N;
4      int _LOG;
5      vector <T> _A;
6      vector < vector <T> > ST;
7      vector <int> Log;
8      F func;
9
10 public :
11     SparseTable() = default;
12
13     template <class iter>
14     SparseTable(iter _begin, iter _end, const F _func = less <T> ()) : func(_func) {
15         _N = distance(_begin, _end);
16
17         Log.assign(_N + 1, 0);
18         for(int i = 2; i <= _N; ++i)
19             Log[i] = Log[i >> 1] + 1;
20
21         _LOG = Log[_N];
22
23         _A.assign(_N + 1, 0);
24         ST.assign(_N + 1, vector <T> (_LOG + 1, 0));
25
26         _typeof(_begin) i = _begin;
27         for(int j = 1; i != _end; ++i, ++j)
28             _A[j] = *i;
29
30         build();
31     }
32
33     void build() {
34         for(int i = 1; i <= _N; ++i)
35             ST[i][0] = i;
36
37         for(int j = 1, k, d; j <= _LOG; ++j) { // the two nested loops below have overall time complexity
38             // = O(n log n)
39             k = (1 << j);
40             d = (k >> 1);
41
42             for(int i = 1; i + k - 1 <= _N; ++i) {
43 T const& x = ST[i][j - 1]; // starting subarray at index = i with length = 2^(j - 1)
44 T const& y = ST[i + d][j - 1]; // starting subarray at index = i + d with length = 2^(j - 1)
45
46             ST[i][j] = func(_A[x], _A[y]) ? x : y;
47         }
48     }
49
50     T query(int l, int r) { // this query is O(1)
51         int d = r - l + 1;
52         T const& x = ST[l][Log[d]];
53         T const& y = ST[l + d - (1 << Log[d])][Log[d]];
54         return func(_A[x], _A[y]) ? x : y;
55     }
56 }
57 };

```

### 3.5 Sparse table (rsq)

```

1  template <class T, class F = function <T(const T &, const T &)>>
2  class SparseTable {
3      int _N;
4      int _LOG;
5      vector <T> _A;
6      vector < vector <T> > ST;
7      vector <int> Log;

```

```

8      F func;
9
10 public :
11     SparseTable() = default;
12
13     template <class iter>
14     SparseTable(iter _begin, iter _end, F _func = [] (T a, T b) { return a + b; }) : func(_func) {
15         _N = distance(_begin, _end);
16
17         Log.assign(_N + 1, 0);
18         for(int i = 2; i <= _N; ++i)
19             Log[i] = Log[i >> 1] + 1;
20
21         _LOG = Log[_N];
22
23         _A.assign(_N + 1, 0);
24         ST.assign(_N + 1, vector <T> (_LOG + 1, 0));
25
26         _typeof(_begin) i = _begin;
27         for(int j = 1; i != _end; ++i, ++j)
28             _A[j] = *i;
29
30         build();
31     }
32
33     void build() {
34         for(int i = 1; i <= _N; ++i)
35             ST[i][0] = _A[i];
36
37         for(int j = 1, k, d; j <= _LOG; ++j) {
38             k = (1 << j);
39             d = (k >> 1);
40
41             for(int i = 1; i + k - 1 <= _N; ++i) {
42 T const& x = ST[i][j - 1]; // starting subarray at index = i with length = 2^(j - 1)
43 T const& y = ST[i + d][j - 1]; // starting subarray at index = i + d with length = 2^(j - 1)
44
45             ST[i][j] = func(x, y);
46         }
47     }
48
49     T query(int l, int r) {
50         int d = r - l + 1;
51         T ret = 0;
52
53         for(int i = 1; d; i += lastBit(d), d -= lastBit(d))
54             ret = func(ret, ST[i][Log[lastBit(d)]]);
55
56         return ret;
57     }
58
59     int lastBit(int a) {
60         return (a & -a);
61     }
62 }
63 };

```

### 3.6 Merge sort

```

1  /**
2   * Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time
3   * complexity can be expressed as following recurrence relation.
4
5   T(n) = 2T(n/2) + theta(n)
6
7   The above recurrence can be solved either using the Recurrence Tree method or the Master method. It
8   falls in case II of Master Method and the solution of the recurrence is theta(nLogn). Time
9   complexity of Merge Sort is theta(nLogn) in all 3 cases (worst, average and best) as merge
10  sort always divides the array into two halves and takes linear time to merge two halves.
11
12  Auxiliary Space: O(n)
13  Algorithmic Paradigm: Divide and Conquer
14  Sorting In Place: No in a typical implementation
15  Count Inversion of array: yes
16  https://discuss.codechef.com/t/iit15-editorial/4427
17  Stable: Yes
18
19  **/
20
21 ll inversions;
22
23 template <class T>
24 void merge(T localArr [], int l, int mid, int r) {
25     int l_size = mid - l + 1;
26     int r_size = r - mid;
27
28     T L[l_size + 1];
29     T R[r_size + 1];

```

```

25 for(int i = 0; i < l_size; ++i) L[i] = localArr[i + 1];
26 for(int i = 0; i < r_size; ++i) R[i] = localArr[i + mid + 1];
27
28 T Mx;
29 if(sizeof(T) == 4) Mx = INT_MAX;
30 else Mx = LONG_MAX;
31
32 L[l_size] = R[r_size] = Mx;
33
34 for(int k = 1, i = 0, j = 0; k <= r; ++k)
35     if(L[i] <= R[j])
36         localArr[k] = L[i], i += (L[i] != Mx);
37     else
38         localArr[k] = R[j], j += (R[j] != Mx), inversions += l_size - i;
39 }
40
41 template <class T>
42 void merge_sort(T localArr [], int l, int r) {
43     if(r - l)
44     {
45         int mid = (l + r) >> 1;
46         merge_sort(localArr, l, mid);
47         merge_sort(localArr, mid + 1, r);
48         merge(localArr, l, mid, r);
49     }
50 }
51
52 template <class T>
53 void merge_sort(T _begin, T _end) {
54     const int sz = _end - _begin;
55     __typeof(*_begin) localArray[sz];
56
57     __typeof(_begin) k = _begin;
58     for(int i = 0; k != _end; ++i, ++k)
59         localArray[i] = *k;
60
61     merge_sort(localArray, 0, sz - 1);
62
63     k = _begin;
64     for(int i = 0; k != _end; ++i, ++k)
65         *k = localArray[i];
66 }

```

## 3.7 Selection sort

```

1 template <class T>
2 void selection_sort(T _begin, T _end, int round) {
3     const int sz = _end - _begin;
4     int localArray[sz];
5
6     __typeof(_begin) k = _begin;
7     for(int i = 0; k != _end; ++i, ++k)
8         localArray[i] = *k;
9
10    int MnInx;
11    round = min(sz, round);
12    for(int i = 0; i < round; ++i) {
13        MnInx = i;
14        for(int j = i + 1; j < sz; ++j)
15            if(localArray[j] < localArray[MnInx])
16                MnInx = j;
17        swap(localArray[MnInx], localArray[i]);
18    }
19
20    k = _begin;
21    for(int i = 0; k != _end; ++i, ++k)
22        *k = localArray[i];
23 }

```

## 3.8 Bubble sort

```

1 /**
2     Bubble sort consists of  $n$  rounds. On each round, the algorithm iterates
3     through the elements of the array. Whenever two consecutive elements are found
4     that are not in correct order, the algorithm swaps them. The algorithm can be
5     implemented as follows:
6 **/
7
8 template <class T>
9 void bubble_sort(T _begin, T _end, int round) {
10     const int sz = _end - _begin;
11     int localArray[sz];

```

```

12     __typeof(_begin) k = _begin;
13     for(int j = 0; k != _end; ++k, ++j)
14         localArray[j] = *k;
15
16     round = min(round, sz);
17     for(int i = 0; i < round; ++i) /*  $n$  rounds ->  $n_{th}$  element */
18         for(int j = 0; j < sz - 1; ++j) if(localArray[j] > localArray[j + 1])
19             swap(localArray[j], localArray[j + 1]);
20
21     k = _begin;
22     for(int j = 0; k != _end; ++k, ++j)
23         *k = localArray[j];
24 }
25
26 /**
27     After the first round of the algorithm, the largest element will be in the correct
28     position, and in general, after  $k$  rounds, the  $k$  largest elements will be in the
29     correct positions.
30 **/

```

# 4 Mathematics

## 4.1 Euler totient function

```

1 /**
2     Constraints:
3     1 <= n <= 1e7
4     2 <= a <= 1014
5
6     Time Complexity:
7     linear_sieve takes  $O(n)$ 
8     Phi takes  $O(n / (\ln(n) - 1.08))$ 
9
10    Space Complexity:
11     $O(\text{MaxN} + n / (\ln(n) - 1.08))$ 
12
13    Explanation:
14     $\Phi(n) = n * ((p_1 - 1) / p_1) * ((p_2 - 1) / p_2) * \dots * ((p_k - 1) / p_k)$ 
15     $\Phi(n) = n * (1 - (1 / p_1)) * (1 - (1 / p_2)) * \dots * (1 - (1 / p_k))$ 
16
17    Applications:
18    Eulers theorem:
19     $a^{\Phi(m)} \text{ cong } 1 \pmod{m}$  if  $a$  and  $m$  are relatively prime.
20
21    Fermats little theorem:
22    when  $m$  is a prime:
23     $a^{\{m \text{ minus } 1\}} \text{ cong } 1 \pmod{m}$ 
24
25    As immediate consequence we also get the equivalence:
26     $a^n \text{ cong } a^{\{n \text{ mod } \Phi(m)\}} \pmod{m}$ 
27    This allows computing  $x^n \text{ mod } m$  for very big  $n$ , especially if  $n$  is the result of another
    computation,
28    as it allows to compute  $n$  under a modulo.
29
30 **/
31 int lp[N], Primes[664580], pnx; /** size of Primes =  $n / (\ln(n) - 1.08)$  */
32
33 void linear_sieve(int n) {
34     for (int i = 2; i <= n; ++i) {
35         if (lp[i] == 0) {
36             lp[i] = Primes[pnx++] = i;
37         }
38         for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
39             lp[comp] = Primes[j];
40         }
41     }
42 }
43
44 ll Phi(ll a) { // for Queries
45     ll ret = a, p;
46     for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
47         if (p * p > a) break;
48         if (a % p) continue;
49         ret -= ret / p;
50         while (a % p == 0) a /= p;
51     }
52     if (a > 1) ret -= ret / a;
53     return ret;
54 }

```



## 4.2 Euler phi (sieve)

```

1  /*
2  Constraints:
3  1 <= n <= 1e7
4
5  Time Complexity:
6  Phi_sieve takes O(n * ln(ln(n)))
7
8  Space Complexity:
9  MaxN
10 */
11
12 int EulerPhi[N];
13
14 void Phi_sieve(int n) {
15     for (int i = 1; i <= n; ++i) {
16         EulerPhi[i] = i;
17     }
18     for (int i = 2; i <= n; ++i) {
19         if (EulerPhi[i] == i) {
20             for (int j = i; j <= n; j += i) {
21                 EulerPhi[j] -= EulerPhi[j] / i;
22             }
23         }
24     }
25 }

```

## 4.3 Extended wheel factorization

```

1  /*
2  Constraints:
3  1 <= n <= 1e7
4  2 <= a <= 1e14
5
6  Time Complexity:
7  linear_sieve takes O(n)
8  Factorization takes O(n / (ln(n) - 1.08))
9
10 Space Complexity:
11 O(MaxN + n / (ln(n) - 1.08))
12 */
13
14 int lp[N];
15 int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
16
17 void linear_sieve(int n) {
18     for (int i = 2; i <= n; ++i) {
19         if (lp[i] == 0) {
20             lp[i] = Primes[pnx++] = i;
21         }
22         for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
23             lp[comp] = Primes[j];
24         }
25     }
26 }
27
28 vector<pair<ll, int>> Factorization(ll a) {
29     vector<pair<ll, int>> ret;
30     ll p;
31     for (int i = 0, cnt; i < pnx && (p = Primes[i], true) && p * p <= a; ++i) {
32         if (a % p) continue;
33         cnt = 0;
34         while (a % p == 0) a /= p, ++cnt;
35         ret.emplace_back(p, cnt);
36     }
37     if (a > 1) ret.emplace_back(a, 1);
38     return ret;
39 }

```

## 4.4 Least prime factorization

```

1  /*
2  Constraints:
3  1 <= n <= 1e7
4
5  Time Complexity:
6  linear_sieve takes O(n)
7  Factorization takes O(log(n))
8
9  Space Complexity:

```

```

10 O(MaxN + n / (ln(n) - 1.08))
11 */
12
13 int lp[N];
14 int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
15
16 void linear_sieve(int n) {
17     for (int i = 2; i <= n; ++i) {
18         if (lp[i] == 0) {
19             lp[i] = Primes[pnx++] = i;
20         }
21         for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
22             lp[comp] = Primes[j];
23         }
24     }
25 }
26
27 vector<pair<int, int>> Factorization(int n) {
28     vector<pair<int, int>> ret;
29     while (n > 1) {
30         int p = leastPrime[n], cnt = 0;
31         while (n % p == 0) n /= p, ++cnt;
32         ret.emplace_back(p, cnt);
33     }
34     return ret;
35 }

```

## 4.5 Miller rabin test

```

1 ll ModExp(ll base, ll e, ll mod) {
2     ll result;
3     base %= mod;
4
5     for(result = 1; e; e >= 1ll) {
6         if(e & 1ll)
7             result = ((1l28)result * base) % mod;
8         base = ((1l28)base * base) % mod;
9     }
10    return result;
11 }
12
13 bool CheckComposite(ll n, ll p, ll d, int r) {
14     ll a = ModExp(p, d, n);
15     if(a == 1 || a == n - 1)
16         return false;
17
18     for(int i = 1; i < r; ++i) {
19         a = ((1l28)a * a) % n;
20         if(a == n - 1)
21             return false;
22     }
23     return true;
24 }
25
26 bool Miller(ll n) {
27     if(n < 2) return false;
28
29     int r; ll d;
30     for(r = 0, d = n - 1; (d & 1ll) == 0; d >= 1ll, ++r);
31
32     for(int p : {2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
33         if(n == p)
34             return true;
35         if(CheckComposite(n, p, d, r))
36             return false;
37     }
38     return true;
39 }

```

## 4.6 Mobius function

```

1  /**
2  Constraints:
3  1 <= x <= 1e7
4  2 <= n <= 10^14
5
6  Time Complexity:
7  linear_sieve takes O(x)
8  mobius takes O(n / (ln(n) - 1.08))
9
10 Space Complexity:
11 O(MaxN + n / (ln(n) - 1.08))

```

```

12  */
13
14  int lp[N], Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
15
16  void linear_sieve(int x) {
17      for (int i = 2; i <= x; ++i) {
18          if (lp[i] == 0) {
19              lp[i] = Primes[pnx++] = i;
20          }
21          for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= x; ++j) {
22              lp[comp] = Primes[j];
23          }
24      }
25  }
26
27  int mobius(ll n) {
28      ll p, pp;
29      char mob = 1;
30      for (int i = 0; i < pnx && (p = Primes[i], pp = p * p, true); ++i) {
31          if (pp > n) break;
32          if (n % p) continue;
33          if (n % pp == 0) return 0;
34          n /= p;
35          mob = -mob;
36      }
37      if (n > 1) mob = -mob;
38      return mob;
39  }
40 }

```

## 4.7 Mobius (sieve)

```

1  /*
2   Constraints:
3   1 <= n <= 1e7
4
5   Time Complexity:
6   mu_sieve takes O(n)
7
8   Space Complexity:
9   O(MaxN)
10  */
11
12  int mu[N], lp[N], Primes[78522], pnx;
13
14  void mu_sieve(int n) {
15      mu[1] = 1;
16      fill(mu, mu + N, 1);
17      for (int i = 2; i <= n; ++i) {
18          if (lp[i] == 0) {
19              lp[i] = Primes[pnx++] = i;
20              mu[i] = -1;
21          }
22          for (int j = 0, nxt; j < pnx && Primes[j] <= lp[i] && (nxt = i * Primes[j]) <= n; ++j) {
23              lp[nxt] = Primes[j];
24              mu[nxt] = (lp[i] == Primes[j] ? 0 : -mu[i]);
25          }
26      }
27  }

```

## 4.8 Phi factorial

```

1  /**
2   Constraints:
3   1 <= x <= 1e7
4   2 <= n <= 1e7
5
6   Time Complexity:
7   linear_sieve takes O(x)
8   phi_factorial takes O(n)
9
10  Space Complexity:
11  O(MaxN + n / (ln(n) - 1.08))
12  */
13
14  int lp[N], Primes[664580], pnx; /** number of primes = n / (ln(n) - 1.08) */
15
16  void linear_sieve(int x) {
17      for (int i = 2; i <= x; ++i) {
18          if (lp[i] == 0) {
19              lp[i] = Primes[pnx++] = i;
20          }

```

```

21
22      for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= x; ++j) {
23          lp[comp] = Primes[j];
24      }
25  }
26
27  ll phi_factorial(int n) {
28      ll ret = 1;
29      for (int i = 2; i <= n; ++i) {
30          ret = ret * (lp[i] == i ? i - 1 : i);
31      }
32      return ret;
33  }
34 }

```

## 4.9 Pisano periodic sequence

```

1  /** Algorithm 1: */
2
3  /** Constraints :
4   1 <= n <= CR
5   Where CR stands for your computing resources. In my case,
6   CR = 100,000,000
7   -----
8   The algorithm is constructed around the ideas that a Pisano sequence always starts with 0 and 1,
9   and that this sequence of Fibonacci
10  numbers taken modulo n can be constructed for each number by adding the previous remainders and
11  taking into account the modulo n.
12  -----
13
14  definition:
15  The sequence of Fibonacci numbers {F_n} is periodic modulo any modulus m (Wall 1960), and the
16  period (mod m)
17  is the known as the Pisano period pi(m) (Wrench 1969). For m=1, 2, ..., the values of pi(m) are
18  1, 3, 8, 6, 20, 24, 16, 12, 24, 60, 10, ... (OEIS A001175).
19
20  Since pi(10)=60, the last digit of F_n repeats with period 60, as first noted by Lagrange in 1774
21  (Livio 2002, p. 105).
22  The last two digits repeat with a period of 300, and the last three with a period of 1500.
23  In 1963, Geller found that the last four digits have a period of 15000 and the last five a period
24  of 150000.
25  Jarden subsequently showed that for d>=3, the last d digits have a period of 15*10^(d minus 1) (
26  Livio 2002, pp. 105-106).
27  The sequence of Pisano periods for n=1, 10, 100, 1000, ... are therefore 60, 300, 1500, 15000,
28  150000, 1500000, ... (OEIS A096363).
29  pi(m) is even if m>2 (Wall 1960). pi(m)=m iff m=24*5^k for some integer k>1 (Fulton and Morris
30  1969, Wrench 1969).
31
32  resources :
33  1. https://webbox.lafayette.edu/~reiterc/nt/qf_fib_ec_preprint.pdf
34  2. https://www.youtube.com/watch?v=Nu-lW-Ifyec&ab_channel=Numberphile
35  3. http://webspace.ship.edu/msrenault/fibonacci/fib.htm
36  4. http://www.theoremoftheday.org/Binomial/PeriodicFib/TotDPeriodic.pdf
37  5. http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibmaths.html#fibmod
38  6. http://webspace.ship.edu/msrenault/fibonacci/fibthesis.pdf
39  7. https://www.fq.math.ca/Scanned/1-2/vinson.pdf
40  8. https://www.fq.math.ca/Scanned/1-2/vinson.pdf
41  9. https://www.fq.math.ca/Scanned/1-2/vinson.pdf
42  */
43
44  vector<int> pisano_periodic_sequence(int n) {
45      vector<int> period;
46
47      int current = 0, next = 1;
48      period.push_back(current);
49
50      if (n < 2) return period;
51      current = (next += current) - current;
52
53      while (current != 0 || next != 1) {
54          period.push_back(current);
55          current = current + next >= n ? (next += current - n) + (n - current) : (next += current) -
56          current;
57      }
58      return period;
59  }
60
61  /**
62   ****
63   /** Algorithm 2: */
64
65  /** constraints:
66   1 <= n <= 10^18]
67   -----
68  problem statement (https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&page
69  =show_problem&problem=4479):
70  For any integer n, the sequence of Fibonacci numbers F_i taken mod n is periodic.
71  define K(n) = the length of the period of the Fibonacci sequence reduced mod n.
72  The task is to print the length of the period of this sequence K(n).
73  -----

```

```

61 Definition:
62 The Pisano period  $\pi(n)$  is a multiplicative function, that is if  $a$  and  $b$  are coprime than  $\pi(ab) = \pi(a) * \pi(b)$ .
63 So we need only concern ourselves with the value of  $\pi(p^k)$  for prime  $p$ .
64 (Factoring even a large number is still better than brute force periodicity search.)
65
66 It is hypothesized that  $\pi(p^k) = \pi(p^{k-1}) * \pi(p)$  and since no counterexamples are known to exist,
67 you might as well use that in your algorithm.
68
69 So, how to calculate  $\pi(p)$  efficiently? There are two special cases and two general cases
70
71  $\pi(2^k) = 3 * 2^{k-1}$ 
72  $\pi(5^k) = 4 * 5^{k-1}$ 
73
74 If  $p \bmod 10 = 1$  or  $p \bmod 10 = 9$  then  $\pi(p) \mid p-1$ 
75 If  $p \bmod 10 = 3$  or  $p \bmod 10 = 7$  then  $\pi(p) \mid 2 * (p+1)$ , and by an odd divisor too.
76
77 The last two statements give us a relatively small number of cases to try (after factoring  $p-1$ 
78 or  $2*(p+1)$ .)
79 Now use your favorite formula to calculate large values of the Fibonacci numbers  $F(x) \bmod p$ .
80 [See Michal answer to What is a fast algorithm to find the remainder of the division of a huge
81 Fibonacci number by some big integer?](https://www.quora.com/Whats-a-fast-algorithm-to-find-the-remainder-of-the-division-of-a-huge-Fibonacci-number-by-some-big-integer/answer/Michal-Fori%C5%A1ek)
82
83 To test a candidate period  $R$ , calculate  $F(R) \bmod p$  and  $F(R+1) \bmod p$ .
84 If these are equal to  $F(0) = 0$  and  $F(1) = 1$ , then  $\pi(p) \mid R$ .
85
86 It might be that  $p-1$  or  $2*(p+1)$  have a lot of divisors, but we dont need to try them all.
87 Suppose  $q^k \mid R$  for some prime  $q$ .
88 Then test  $R/q$ . If that doesnt produce a cycle, then  $\pi(p)$  must have factor  $q^k$ ,
89 and we can leave it in and go on to other factors.
90 Otherwise, we can use  $R/q$  as our new starting point and repeat the process.
91 Thus we have to do a number of checks proportional to  $\Omega(2*(p+1))$ , not  $d(2*(p+1))$ .
92
93 -----
94 Donald Wall proved several other properties, some of which you may find interesting:
95 If  $m > 2$ ,  $k(m)$  is even.
96 For any even integer  $n > 2$ , there exists  $m$  such that  $k(m) = n$ .
97  $k(m) \leq (m^2 - 1)$ 
98  $k(2^n) = 3 * 2^{n-1}$ 
99  $k(5^n) = 4 * 5^{n-1}$ 
100 If  $n > 2$ ,  $k(10^n) = 15 * 10^{n-1}$ 
101  $k(2 * 5^n) = 6n$ 
102
103 **/
104
105 #pragma GCC optimize ("Ofast")
106
107 #include <bits/stdc++.h>
108
109 #define endl '\n'
110
111 using namespace std;
112
113 typedef long long ll;
114 typedef __int128 i128;
115 typedef __int128_t u128;
116
117 template <class T>
118 using matrix = vector < vector <T> >;
119
120 template <class T> string to_string(T x) {
121     int sn = 1;
122     if(x < 0) sn = -1, x *= sn;
123     string s = "";
124     do {
125         s = "0123456789"[x % 10] + s, x /= 10;
126     } while(x);
127     return (sn == -1 ? "-" : "") + s;
128 }
129
130 auto str_to_int(string x) {
131     u128 ret = (x[0] == '-' ? 0 : x[0] - '0');
132     for(int i = 1; i < (int)x.size(); ++i) ret = ret * 10 + (x[i] - '0');
133     return (x[0] == '-' ? -1 * (i128)ret : ret);
134 }
135
136 istream & operator >> (istream & in, i128 & i) noexcept {
137     string s;
138     in >> s;
139     i = str_to_int(s);
140     return in;
141 }
142
143 ostream & operator << (ostream & os, const i128 i) noexcept {
144     os << to_string(i);
145     return os;
146 }
147
148 void Fast() {
149     cin.sync_with_stdio(0);
150     cin.tie(0);
151     cout.tie(0);
152 }

```

```

148 }
149
150 ll n;
151 vector <int> primes;
152 matrix <ll> fibMatrix = {{1, 1},
153 {1, 0}};
154 };
155
156 i128 gcd(i128 a, i128 b) {
157     while (a && b)
158         a > b ? a %= b : b %= a;
159     return a + b;
160 }
161
162 i128 lcm(i128 a, i128 b) {
163     return a / gcd(a, b) * b;
164 }
165
166 vector < array <ll, 2> > factorize(ll x) {
167     vector < array <ll, 2> > ret;
168     for(int i = 0; i11 * primes[i] * primes[i] <= x; ++i) {
169         if(x % primes[i]) continue;
170
171         int cnt = 0;
172         while (x % primes[i] == 0) {
173             cnt++;
174             x /= primes[i];
175         }
176         ret.push_back({primes[i], cnt});
177     }
178
179     if(x > 1) ret.push_back({x, 1});
180     return ret;
181 }
182
183 matrix <ll> MatMul(matrix <ll> A, matrix <ll> B, ll mod) {
184     int ra = A.size(), cb = B[0].size(), ca = A[0].size();
185     matrix <i128> C(ra, vector <i128> (cb));
186
187     for(int i = 0; i < ra; ++i)
188         for (int j = 0; j < cb; ++j) {
189             C[i][j] = 0;
190             for(int k = 0; k < ca; ++k)
191                 C[i][j] = (C[i][j] + (i128)A[i][k] * B[k][j]) % mod;
192         }
193
194     matrix <ll> ret(ra, vector <ll> (cb));
195     for(int i = 0; i < ra; ++i)
196         for (int j = 0; j < cb; ++j)
197             ret[i][j] = C[i][j] % mod;
198
199     return ret;
200 }
201
202 matrix <ll> MatPow(matrix <ll> A, ll p, ll mod) {
203     int r = A.size(), c = A[0].size();
204     assert(r == c && p);
205     matrix <ll> result = A;
206     p--;
207
208     while(p) {
209         if(p & 1ll) result = MatMul(result, A, mod);
210         A = MatMul(A, A, mod);
211         p >>= 1ll;
212     }
213     return result;
214 }
215
216 i128 ModExp(i128 a, ll p) {
217     i128 result = 1;
218     while(p) {
219         if(p & 1ll) result = result * a;
220         a *= a;
221         p >>= 1ll;
222     }
223     return result;
224 }
225
226 ll nthFib(ll n, ll mod) {
227     return MatPow(fibMatrix, n, mod)[0][1];
228 }
229
230 bool is_period(ll n, ll mod) {
231     return nthFib(n, mod) == 0 && nthFib(n + 1, mod) == 1;
232 }
233
234 ll solver(ll x, ll mod) {
235     vector < array <ll, 2> > factors = factorize(x);
236     for(int i = 0; i < (int)factors.size(); ++i) {
237         while(x % factors[i][0] == 0 && is_period(x / factors[i][0], mod))
238             x /= factors[i][0];
239     }

```

```

240     return x;
241 }
242
243 ll pisano_prime(ll val) {
244     if(val == 2) return 3;
245     if(val == 5) return 20;
246     if(val % 10 == 1 || val % 10 == 9)
247         return solver(val - 1, val);
248
249     return solver(2 * (val + 1), val);
250 }
251
252 const int N = 1e7 + 9;
253 bitset<N> isPrime;
254
255 void Precomputation_Sieve() {
256     isPrime.set();
257     int _sqrt = sqrtl(N);
258
259     for(int i = 5; i <= _sqrt; i += 6) {
260         if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
261         i += 2;
262         if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
263         i -= 2;
264     }
265 }
266
267 vector<int> Primes(int n) {
268     vector<int> _Primes;
269
270     if(n >= 2) _Primes.push_back(2);
271     if(n >= 3) _Primes.push_back(3);
272
273     for (int i = 5; i <= n; i += 6) {
274         if(isPrime[i]) _Primes.push_back(i);
275         i += 2;
276         if(isPrime[i]) _Primes.push_back(i);
277         i -= 2;
278     }
279     return _Primes;
280 }
281
282 void initialize()
283 {
284     Precomputation_Sieve();
285     primes = Primes(N);
286 }
287
288 void Solve() {
289     initialize();
290     cin >> n;
291     vector<array<ll, 2>> factors = factorize(n);
292
293     ll28 ans = 1;
294     for (int i = 0; i < (int)factors.size(); ++i) {
295         ans = lcm(ans, (ll28)pisano_prime(factors[i][0]) * ModExp(factors[i][0], factors[i][1] - 1));
296     }
297     cout << ans << endl;
298 }

```

## 4.10 Simple sieve

```

1  /**
2   constraints:
3   1 <= n <= 1e8
4
5   Time complexity:
6   O(n * ln(ln(sqrt(n))) + n)
7
8   Space Complexity:
9   O(MaxN + n / (ln(n) - 1.08))
10  */
11
12 bool isPrime[N];
13 int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
14
15 void sieve(int x) {
16     int basis[3] = {2, 3, 5};
17     int wheel[8] = {7, 11, 13, 17, 19, 23, 29, 1};
18     int inc[8] = {4, 2, 4, 2, 4, 6, 2, 6};
19     int inx[31];
20
21     memset(inx, 0, sizeof(inx));
22     memset(isPrime, true, sizeof(isPrime));
23
24     for (int p : basis) if (x > p) Primes[pnx++] = p;
25     for (int i = 0; i < 8; ++i) inx[wheel[i]] = i;

```

```

26
27 int c = 0;
28 for (int i = 7; i <= x; i += inc[c++]) {
29     if (isPrime[i]) {
30         Primes[pnx++] = i;
31         int d = inx[i % 30];
32         for (ll j = i * 111 + i; j <= x; j += i * inc[d++]) {
33             isPrime[j] = false;
34             if (d == 8) d = 0;
35         }
36     }
37     if (c == 8) c = 0;
38 }
39 }

```

## 4.11 wheel sieve

```

1  /**
2   Constraints:
3   1 <= n <= 1e9
4   2 <= x <= 9700000
5
6   Time Complexity:
7   wheel_sieve takes O(n / ln(ln(n)))
8   coPrimes takes O(x * ln(ln(x)))
9
10  Space Complexity:
11  O(MaxN / 32 + n / (ln(n) - 1.08) + x)
12  */
13
14 bitset<N> isPrime;
15 int inx[30100];
16 int Primes[50908031], pnx; /** size of Primes = n / (ln(n) - 1.08) */
17
18 vector<int> coPrimes(int x) {
19     int basis[5] = {3, 5, 7, 11, 13};
20
21     vector<int> ret;
22     bitset<30100> isCoprime;
23     isCoprime.set();
24
25     for (int b : basis)
26         for (int d = b + b; d <= x; d += b << 1)
27             isCoprime.reset(d);
28
29     for (int i = 17; i <= x; i += 2)
30         if (isCoprime[i]) ret.push_back(i);
31
32     ret.push_back(x + 1);
33     ret.push_back(x + 17);
34     return ret;
35 }
36
37 void wheel_sieve(int n) {
38     int basis[6] = {2, 3, 5, 7, 11, 13};
39     vector<int> wheel = coPrimes(2 * 3 * 5 * 7 * 11 * 13);
40     int sz = wheel.size();
41
42     for (int k = 0; k < sz; ++k)
43         inx[wheel[k]] = k;
44
45     isPrime.set();
46     inx[1] = sz - 2;
47     int inc[sz - 1];
48
49     for (int i = 1; i < sz; ++i)
50         inc[i - 1] = wheel[i] - wheel[i - 1];
51
52     for (int p : basis) {
53         if (n >= p)
54             Primes[pnx++] = p;
55     }
56
57     int c = 0;
58     for (ll i = 17; i <= n; i += inc[c++]) {
59         if (isPrime[i]) {
60             Primes[pnx++] = i;
61             int d = inx[i % 30030];
62             for (ll j = i * i; j <= n; j += i * inc[d++]) {
63                 isPrime.reset(j);
64                 if (d == sz - 1) d = 0;
65             }
66         }
67         if (c == sz - 1) c = 0;
68     }
69 }

```

## 4.12 Linear sieve

```

1  /**
2   Constraints:
3   1 <= n <= 1e7
4
5   Time Complexity:
6   linear_sieve takes O(n)
7
8   Space Complexity:
9   O(MaxN + n / (ln(n) - 1.08))
10  */
11
12 int lp[N];
13 int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
14
15 void linear_sieve(int n) {
16     for (int i = 2; i <= n; ++i) {
17         if (lp[i] == 0) {
18             lp[i] = Primes[pnx++] = i;
19         }
20         for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
21             lp[comp] = Primes[j];
22         }
23     }
24 }

```

## 4.13 Segmented sieve

```

1  /**
2   constraints:
3   1 <= l, r <= 1e{14}
4   1 <= r - l + 1 <= 1e7
5   2 <= x <= 1e7
6
7   Time complexity:
8   segmented_sieve takes O((r - l + 1) * ln(ln(r)))
9   linear_sieve takes O(n)
10
11   Space Complexity:
12   O(2 * MaxN + n / (ln(n) - 1.08))
13  */
14
15 int lp[N];
16 int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
17 bool isPrime[N];
18
19 void linear_sieve(int n) {
20     for (int i = 2; i <= n; ++i) {
21         if (lp[i] == 0) {
22             lp[i] = Primes[pnx++] = i;
23         }
24         for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
25             lp[comp] = Primes[j];
26         }
27     }
28 }
29
30 vector<ll> segmented_sieve(ll l, ll r) {
31     l += 1 == 1;
32     int limit = r - l + 1;
33     vector<ll> ret;
34     memset(isPrime, true, sizeof(isPrime));
35
36     ll p;
37     for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
38         for (ll j = max(p * p, (l + p - 1) / p * p); j <= r; j += p)
39             isPrime[j - l] = false;
40     }
41
42     for (int i = 0; i < limit; ++i)
43         if (isPrime[i])
44             ret.emplace_back(i + l);
45     return ret;
46 }

```

## 4.14 The stable marriage problem

```

1  const int N = 1e3 + 9, M = 1e3 + 9, oo = 0x3f3f3f3f;
2  queue<int> Q;
3
4  int husband[N], wife[N], Next[N], order[N][N], pref[N][N], n, v;
5
6  void _clear() {
7      memset(wife, 0, sizeof(wife[0]) * (n + 2));
8      memset(husband, 0, sizeof(husband[0]) * (n + 2));
9      memset(Next, 0, sizeof(Next[0]) * (n + 2));
10 }
11
12 void engage(int man, int woman) {
13     int exWife = wife[man];
14     wife[man] = woman;
15     husband[woman] = man;
16
17     if (exWife)
18         Q.push(exWife);
19 }
20
21 void Solve() {
22     _clear();
23     cin >> n;
24
25     for (int i = 1; i <= n; ++i)
26         for (int j = 1; j <= n; ++j)
27             cin >> pref[i][j];
28
29     for (int i = 1; i <= n; ++i)
30         for (int j = 1; j <= n; ++j) {
31             cin >> v;
32             order[i][v] = j;
33         }
34
35     for (int i = 1; i <= n; ++i)
36         Q.push(i);
37
38     int man, woman;
39     while (Q.size()) {
40         woman = Q.front(); Q.pop();
41         man = pref[woman][++Next[woman]];
42
43         if (!wife[man] || order[man][woman] < order[man][wife[man]])
44             engage(man, woman);
45         else
46             Q.push(woman);
47     }
48
49     for (int i = 1; i <= n; ++i)
50         cout << husband[i] << endl;
51 }

```

## 5 String processing

### 5.1 Trie

```

1  class Trie {
2  private:
3      Trie* children[26]; // Pointer = 8 Byte; 8*26 = 208 Byte
4      int prefixes, words; // 8 Byte
5      bool iseow; // 1 Byte
6      char cur_letter; // 1 Byte
7      vector<string> lex;
8      priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int, string>>>
9          occurrence; // small at top
10
11 public:
12     Trie(char lett = '\0') {
13         memset(children, 0, sizeof(children));
14         prefixes = words = 0;
15         iseow = false;
16         cur_letter = lett;
17     }
18
19     void insert(string &str) { // O(l)
20         Trie* cur = this;
21         int inx, strsz = str.size();
22         for (int i = 0; i < strsz; ++i) {
23             inx = str[i] - 'a';
24             if (cur->children[inx] == nullptr)
25                 cur->children[inx] = new Trie(str[i]);
26
27             cur = cur->children[inx];
28             cur->prefixs++;
29         }
30         if (str[strsz - 1] != '\0')
31             cur->words++;
32     }
33
34     void print() {
35         priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int, string>>>
36             q;
37         for (int i = 0; i < 26; ++i)
38             if (children[i])
39                 q.push({children[i]->prefixs, children[i]->lex[i]});
40         while (!q.empty()) {
41             pair<int, string> p = q.top(); q.pop();
42             cout << p.second << " " << p.first << endl;
43         }
44     }
45 };

```

```

28     }
29     cur->iseow = true;
30     cur->words++;
31 }
32
33 int search_word(string &str) { // O(1)
34     Trie* cur = this;
35     int inx, strsz = str.size();
36     for(int i = 0; i < strsz; ++i) {
37         inx = str[i] - 'a';
38         if(cur->children[inx] == nullptr) {
39             return 0;
40         }
41         cur = cur->children[inx];
42     }
43     return cur->words;
44 }
45
46 int search_prefix(string &str) { // O(1)
47     Trie* cur = this;
48     int inx = 0, strsz = str.size();
49     for(int i = 0; i < strsz; ++i) {
50         inx = str[i] - 'a';
51         if(cur->children[inx] == nullptr) {
52             return 0;
53         }
54         cur = cur->children[inx];
55     }
56     return cur->prefixs;
57 }
58
59 bool erase(string &str) {
60     if(!search_word(str))
61         return false;
62
63     Trie* cur = this;
64     int inx, strsz = str.size();
65     for(int i = 0; i < strsz; ++i) {
66         inx = str[i] - 'a';
67         if(--cur->children[inx]->prefixs == 0) {
68             cur->children[inx] = nullptr;
69             return true;
70         }
71         cur = cur->children[inx];
72     }
73     if(--cur->words == 0) {
74         cur->iseow = false;
75     }
76     return true;
77 }
78
79 private:
80 void dfs(Trie* node, string s) { // lex order dfs -> traverse all the strings starting from root
81     if(node->iseow) {
82         lex.emplace_back(s);
83     }
84
85     for(int j = 0; j < 26; ++j)
86         if(node->children[j] != nullptr) {
87             dfs(node->children[j], s + string(1, node->children[j]->cur_letter));
88         }
89 }
90
91 void dfs2(Trie* node, string s) { // autocomplete dfs -> traverse all the strings starting from the
92     // end of the given prefix
93     if(node->iseow) {
94         if(occurrence.size() < 10) {
95             occurrence.push(make_pair(node->words, s));
96         } else {
97             if(node->words > occurrence.top().first) {
98                 occurrence.pop();
99                 occurrence.push(make_pair(node->words, s));
100             }
101         }
102
103         for(int i = 0; i < 26; ++i) if(node->children[i] != nullptr) {
104             dfs2(node->children[i], s + string(1, node->children[i]->cur_letter));
105         }
106     }
107
108 public:
109 vector<string> lex_order() { // all strings in lexicographical order
110     lex.clear();
111     Trie* cur = this;
112     for(int i = 0; i < 26; ++i) if(cur->children[i] != nullptr) {
113         dfs(cur->children[i], string(1, cur->children[i]->cur_letter));
114     }
115     return lex;
116 }
117
118 void autocomplete(string &pref) { // suggest top ten words with max frequency

```

```

119     if(!search_prefix(pref))
120         return;
121
122     Trie* cur = this;
123     int inx, presz = pref.size();
124     for(int i = 0; i < presz; ++i) {
125         inx = pref[i] - 'a';
126         cur = cur->children[inx];
127     }
128
129     for(int i = 0; i < 26; ++i) if(cur->children[i] != nullptr) {
130         dfs2(cur->children[i], string(1, cur->children[i]->cur_letter));
131     }
132
133     vector<string> st;
134     while(!occurrence.empty()) {
135         st.emplace_back(pref + occurrence.top().second);
136         occurrence.pop();
137     }
138     if(cur->iseow) {
139         st.emplace_back(pref);
140     }
141     while(!st.empty()) {
142         cout << st.back() << endl;
143         st.pop_back();
144     }
145 }
146 };

```

## 5.2 Knuth Morris Pratt (kmp)

```

1  /**
2   * KMP (Knuth-Morris-Pratt) Algorithm
3   * Longest Prefix
4   * proper prefix = all prefixes except the whole string
5   * propre suffix = all suffixes except the whole string
6   * Prefix Function = Failure Function
7   * Given String P of len m, Find F[m];
8   * let t = P[0...i]
9   * f[i] = length of the longest proper prefix of t that is suffix of t
10  * calculating i different ways
11  * match the pattern against itself
12  * O(m) for failure function
13  * O(n) for KMP
14  */
15
16 vector<int> LongestPrefix(string &p) {
17     int psz = p.size();
18     vector<int> longest_prefix(psz, 0);
19
20     for(int i = 1, k = 0; i < psz; ++i) {
21         while(k && p[k] != p[i]) k = longest_prefix[k - 1];
22         longest_prefix[i] = (p[k] == p[i] ? ++k : k);
23     }
24     return longest_prefix;
25 }
26
27 vector<int> KMP(string &s, string &p) {
28     int ssz = s.size(), psz = p.size();
29
30     vector<int> longest_prefix = LongestPrefix(p), matches;
31
32     for(int i = 0, k = 0; i < ssz; ++i) {
33         while(k && p[k] != s[i]) k = longest_prefix[k - 1]; // Fail go back
34         k += (p[k] == s[i]);
35
36         if(k == psz) {
37             matches.emplace_back(i - psz + 1);
38             k = longest_prefix[k - 1]; // fail safe and find another pattern
39         }
40     }
41     return matches;
42 }

```

## 6 Geometry

### 6.1 Point

```

1  /**

```

```

2  notes:
3  EPS = 1e-9
4  -----
5  Integers | Doubles |
6  -----
7  a == b | fabs(a - b) < EPS |
8  a <= b | a < b + EPS |
9  a >= b | a + EPS > b |
10 a < b | a + EPS < b |
11 a > b | a > b + EPS |
12 x >= 0.0 | x > -EPS |
13 x <= 0.0 | x < EPS |
14 -----
15 **/
16
17 class point {
18 public :
19     ld x, y;
20
21     point() = default;
22     point(ld _x, ld _y) : x(_x), y(_y) {}
23
24     bool operator < (point other) const {
25         if(fabs(x - other.x) > EPS) // if(x != other.x)
26             return x < other.x;
27         return y < other.y;
28     }
29
30     bool operator == (point other) const {
31         return ((fabs(x - other.x) < EPS) && (fabs(y - other.y) < EPS)); // " < EPS " equal to " == zero "
32     }
33
34     bool operator > (point other) const {
35         if(fabs(x - other.x) > EPS)
36             return x > other.x;
37         return y > other.y;
38     }
39
40     ld dist(point other) { // Euclidean distance
41         ld dx = this->x - other.x;
42         ld dy = this->y - other.y;
43         return sqrtl(dx * dx + dy * dy);
44     }
45
46     ld DEG_to_RAD(ld theta) {
47         return theta * PI / 180.0;
48     }
49
50     ld RAD_to_DEG(ld theta) {
51         return theta * 180.0 / PI;
52     }
53
54     point rotate(ld theta) {
55         ld rad = DEG_to_RAD(theta);
56         return point(cos(theta) * x - sin(theta) * y,
57                     sin(theta) * x + cos(theta) * y);
58     }
59 };

```

```

22 ret.push_back({sr, sc});
23 reverse(ret.begin(), ret.end());
24 return ret;
25 }
26
27 bool valid(int r, int c) {
28     return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] != '%';
29 }
30
31 /** admissible heuristic **/
32 int manhattanDistance(int x1, int y1, int x2, int y2) {
33     return (abs(x1 - x2) + abs(y1 - y2));
34 }
35
36 int Astar(int sr, int sc, int tr, int tc) {
37     memset(dis, 0x3f, sizeof(dis));
38     memset(Par, -1, sizeof(Par));
39
40     priority_queue <tuple <int, int, int> > Q;
41
42     dis[sr][sc] = 0;
43     Q.push({-manhattanDistance(sr, sc, tr, tc), sr, sc});
44
45     int hcost, r, c, nr, nc;
46     while(Q.size()) {
47         tie(hcost, r, c) = Q.top(); Q.pop();
48         if(r == tr && c == tc) return dis[r][c];
49
50         for(int i = 0; i < 4; ++i) {
51             nr = r + dr[i];
52             nc = c + dc[i];
53
54             if(!valid(nr, nc)) continue;
55
56             if(dis[r][c] + 1 < dis[nr][nc]) {
57                 dis[nr][nc] = dis[r][c] + 1;
58                 Par[nr][nc] = dir[i ^ 2];
59                 Q.push({-dis[nr][nc] - manhattanDistance(nr, nc, tr, tc), nr, nc});
60             }
61         }
62     }
63     return -1;
64 }
65
66 void Solve() {
67     cin >> si >> sj >> ti >> tj >> n >> m;
68     for(int i = 0; i < n; ++i)
69         for(int j = 0; j < m; ++j)
70             cin >> grid[i][j];
71
72     cout << Astar(si, sj, ti, tj) << endl;
73     vector < pair <int, int> > path = restorePath(si, sj, ti, tj);
74
75     for(auto point : path)
76         cout << point.first << " " << point.second << endl;
77 }

```

## 7 More advanced topics

### 7.1 A\* algorithm

```

1  const int dr [] = {-1, 0, 1, 0};
2  const int dc [] = {0, 1, 0, -1};
3  const char dir [] = {'U', 'R', 'D', 'L'};
4  map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};
5
6  const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
7  const ll INF = 0x3f3f3f3f3f3f3f3f;
8
9  char grid[N][N], Par[N][N];
10 int dis[N][N], n, m, si, sj, ti, tj;
11
12 vector < pair <int, int> > restorePath(int sr, int sc, int tr, int tc) {
13     vector < pair <int, int> > ret;
14     if(dis[tr][tc] == oo) return ret;
15
16     for(char i = Par[tr][tc]; (sr ^ tr) || (sc ^ tc); i = Par[tr][tc]) {
17         ret.push_back({tr, tc});
18         tr += dr[inv[i]];
19         tc += dc[inv[i]];
20     }
21 }

```

### 7.2 Mo's algorithm

```

1  const int N = 3e4 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2  const ll INF = 0x3f3f3f3f3f3f3f3f;
3  const int BLK = 256;
4
5  struct query {
6      int l, r, id, blk;
7  };
8
9  query() = default;
10 query(int _l, int _r, int _id) {
11     l = _l;
12     r = _r;
13     id = _id;
14     blk = l / BLK;
15 }
16
17 bool operator < (const query other) const {
18     if(blk ^ other.blk)
19         return blk < other.blk;
20     return (blk & 1) ? r < other.r : r > other.r;
21 }
22 queries[M];
23
24 int res[M], freq[M << 3], cur;
25
26 void add(int id) {
27     cur += (++freq[id] == 1);
28 }

```

```

29 void remove(int id) {
30     cur -= (--freq[id] == 0);
31 }
32 int get_res() {
33     return cur;
34 }
35 }
36 int cur_l, cur_r, l, r, n, q, a[N];
37
38 void Solve() {
39     cin >> n;
40     for(int i = 1; i <= n; ++i) cin >> a[i];
41
42     cin >> q;
43     for(int i = 1; i <= q; ++i) {
44         cin >> l >> r;
45         queries[i] = query(l, r, i);
46     }
47
48     sort(queries + 1, queries + 1 + q);
49
50     cur_l = 1, cur_r = 0; // assign to right invalid index
51     for(int i = 1; i <= q; ++i) {
52         int ql = queries[i].l;
53         int qr = queries[i].r;
54
55         // Add right
56         while(cur_r < qr) add(a[++cur_r]);
57         // Add left
58         while(cur_l > ql) add(a[--cur_l]);
59         // Remove right
60         while(cur_r > qr) remove(a[cur_r--]);
61         // Remove left
62         while(cur_l < ql) remove(a[cur_l++]);
63
64         res[queries[i].id] = get_res();
65     }
66
67     for(int i = 1; i <= q; ++i)
68         cout << res[i] << "\n";
69 }
70 }

```

## 7.3 Square root decomposition

```

1  const int N = 5e5 + 9, M = 1e3 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
2  const ll INF = 0x3f3f3f3f3f3f3f3f;
3  const int BLK = 256;
4
5  int n, q, a[N], type, x, y, z;
6  vector<int> bs[M];
7
8  int query(int l, int r, int val) {
9      int cur_l = l / BLK;
10     int cur_r = r / BLK;
11     int ans = 0;
12
13     if(cur_l == cur_r) {
14         for (int i = l; i <= r; ++i)
15             ans += (a[i] >= val);
16     } else {
17         for(int i = l, _end = (cur_l + 1) * BLK; i < _end; ++i)
18             ans += (a[i] >= val);
19         for(int i = cur_l + 1; i <= cur_r - 1; ++i)
20             ans += bs[i].end() - lower_bound(bs[i].begin(), bs[i].end(), val);
21         for(int i = cur_r * BLK; i <= r; ++i)
22             ans += (a[i] >= val);
23     }
24     return ans;
25 }
26
27 void build() {
28     for(int i = 0; i < n; ++i)
29         bs[i / BLK].emplace_back(a[i]);
30
31     for(int i = 0; i < M; ++i)
32         sort(bs[i].begin(), bs[i].end());
33 }
34
35 void update(int id, int delta) {
36     int pos = lower_bound(bs[id / BLK].begin(), bs[id / BLK].end(), a[id]) - bs[id / BLK].begin();
37     bs[id / BLK][pos] = delta;
38     sort(bs[id / BLK].begin(), bs[id / BLK].end());
39     a[id] = delta;
40 }
41
42 void Solve() {

```

```

43     cin >> n;
44     for(int i = 1; i <= n; ++i) cin >> a[i];
45
46     build();
47     cin >> q;
48     while(q--) {
49         cin >> type >> x >> y;
50         if(type == 0) {
51             cin >> z;
52             cout << query(x, y, z) << endl;
53         }
54         else
55             update(x, y);
56     }
57 }

```

## 8 Miscellaneous

### 8.1 C++ template

```

1  #pragma GCC optimize("Ofast")
2
3  #include <bits/stdc++.h>
4
5  #define endl '\n'
6  #define read(a, n) for(int i = 0; i < n; cin >> a[i++]);
7
8  #define debug(args ...) { \
9      string _s = #args; \
10     replace(_s.begin(), _s.end(), ',', ' '); \
11     stringstream _ss(_s); \
12     istream_iterator<string> _it(_ss); \
13     err(_it, args); \
14 }
15
16 using namespace std;
17
18 using i128 = __int128_t;
19 using i64 = int64_t;
20 using i32 = int32_t;
21
22 using u128 = __uint128_t;
23 using u64 = uint64_t;
24 using u32 = uint32_t;
25 using ld = long double;
26
27 const int N = 2e5 + 9, M = 3e7 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
28 const ld eps = 1e-9;
29
30 void err(istream_iterator<string> it) {}
31 template<typename T, typename ... Args>
32 void err(istream_iterator<string> it, T a, Args ... args) {
33     cerr << *it << " = " << a << endl;
34     err(++it, args ...);
35 }
36
37 void fast() {
38     ios_base::sync_with_stdio(false);
39     cin.tie(nullptr);
40 }
41
42 void file() {
43     freopen("input.in", "r", stdin);
44     freopen("output.out", "w", stdout);
45 }
46
47 void Solve() {
48 }
49
50 int main() {
51     fast();
52     // file();
53
54     int t = 1; // cin >> t;
55     for(int i = 1; i <= t; ++i) {
56         Solve();
57     }
58 }
59 }

```



## 8.2 Double comparison

```

1 bool approximatelyEqual(double a, double b, double epsilon) {
2     return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
3 }
4
5 bool essentiallyEqual(double a, double b, double epsilon) {
6     return fabs(a - b) <= ((fabs(a) > fabs(b) ? fabs(b) : fabs(a)) * epsilon);
7 }
8
9 bool definitelyGreaterThan(double a, double b, double epsilon) {
10    return (a - b) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
11 }
12
13 bool definitelyLessThan(double a, double b, double epsilon) {
14    return (b - a) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
15 }

```

## 8.3 Fast input/output

```

1 /**
2  * Fast Input/Output method for C++:
3  * 1. cin(with sync_with_stdio(false) & cin.tie(nullptr)):
4  *   - int:
5  *     - |n = 5e6| => 420ms
6  *     - |n = 1e7| => 742ms
7  *     - ll:
8  *     - |n = 5e6| => 895ms
9  *
10 * 2. read (using getchar()):
11 *   - int:
12 *     - |n = 5e6| => 173ms
13 *     - |n = 1e7| => 172ms
14 *     - ll:
15 *     - |n = 5e6| => 340ms
16 */
17
18 ll readll () {
19     bool minus = false;
20     unsigned long long result = 0;
21     char ch;
22     ch = getchar();
23
24     while (true) {
25         if (ch == '-') break;
26         if (ch >= '0' && ch <= '9') break;
27         ch = getchar();
28     }
29
30     if (ch == '-') minus = true;
31     else result = ch - '0';
32
33     while (true) {
34         ch = getchar();
35         if (ch < '0' || ch > '9') break;
36         result = result * 10 + (ch - '0');
37     }
38
39     if (minus) return -(ll)result;
40     return result;
41 }
42
43 int readi () {
44     bool minus = false;
45     unsigned int result = 0;
46     char ch;
47     ch = getchar();
48
49     while (true) {
50         if (ch == '-') break;
51         if (ch >= '0' && ch <= '9') break;
52         ch = getchar();
53     }
54
55     if (ch == '-') minus = true;
56     else result = ch - '0';
57
58     while (true) {
59         ch = getchar();
60         if (ch < '0' || ch > '9') break;
61         result = result * 10 + (ch - '0');
62     }
63
64     if (minus) return -(int)result;
65     return result;
66 }

```

## 8.4 Gcd & Lcm

```

1 i64 gcd(i64 a, i64 b) { // binary GCD uses about 60% fewer bit operations
2     if (!a) return b;
3
4     u64 shift = __builtin_ctzll(a | b);
5     a >>= __builtin_ctzll(a);
6
7     while (b) {
8         b >>= __builtin_ctzll(b);
9
10        if (a > b)
11            swap(a, b);
12        b -= a;
13    }
14    return a << shift;
15 }
16
17 i64 lcm(i64 a, i64 b) {
18     return a / gcd(a, b) * b;
19 }

```

## 8.5 Modular calculations

```

1 /*
2  * - It also has important applications in many tasks unrelated to arithmetic, since it can be used
3  *   with any operations that have the property of associativity:
4  */
5 // 1. Modular Exponentiation
6
7 i64 binExp(i64 a, i64 b, i64 p) {
8     i64 res;
9     for (res = 1; b; b >>= 1) {
10         if (b & 1ll)
11             res = res * a % p;
12         a = a * a % p;
13     }
14     return res;
15 }
16
17 // 2. Modular Multiplication
18
19 i64 binMul(i64 a, i64 b, i64 p) {
20     i64 res;
21     a %= p;
22     for (res = 0; b; b >>= 1) {
23         if (b & 1ll)
24             res = (res + a) % p;
25         a = (a + a) % p;
26     }
27     return res;
28 }
29
30 // 3. Modular Multiplicative Inverse
31
32 i64 modInv(i64 b, i64 p) {
33     return binExp(b, p - 2, p); // Guaranteed that p is a Prime Number
34 }

```

## 8.6 Debugging tools

```

1 #define rforeach(_it, c)    for(_typeof((c).rbegin()) _it = (c).rbegin(); _it != (c).rend(); ++_it)
2 #define foreach(_it, c)    for(_typeof((c).begin()) _it = (c).begin(); _it != (c).end(); ++_it)
3 #define all(a)              (a).begin(), (a).end()
4 #define sz(a)                (int)a.size()
5 #define endl                '\n'
6
7 typedef int64_t ll;
8
9 #if __cplusplus >= 201402L
10
11 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
12 template <class T> T rnd_in_rng(T a, T b)
13 { return uniform_int_distribution <ll> (a, b)(rng); /** [a, b] **/ }
14
15 template <typename F, typename S>
16 ostream & operator << (ostream & os, const pair <F, S> & p)

```

```

17 { return os << "(" << p.first << ", " << p.second << ")"; }
18
19 template <typename F, typename S>
20 ostream & operator << (ostream & os, const map <F, S> & _mp)
21 { os << "["; foreach(it, _mp) { if(it != _mp.begin()) os << ", "; os << it->first << " = " << it->
    second; } return os << "]"; }
22
23 template <typename T>
24 ostream & operator << (ostream & os, const vector <T> & _v)
25 { os << "["; foreach(it, _v) { if(it != _v.begin()) os << ", "; os << *it; } return os << "]"; }
26
27 template <typename T>
28 ostream & operator << (ostream & os, const set <T> & _st)
29 { os << "["; foreach(it, _st) { if(it != _st.begin()) os << ", "; os << *it; } return os << "]"; }
30
31 template <typename T, size_t S>
32 ostream & operator << (ostream & os, const array <T, S> & _ar)
33 { os << "["; foreach(it, _ar) { if(it != _ar.begin()) os << ", "; os << *it; } return os << "]"; }
34
35 template <typename T> void write(T _begin, T _end)
36 { for(auto i = _begin; i != _end; ++i) cout << (*i) << ' '; cout << endl; }
37
38 template <typename T> void read(T _begin, T _end)
39 { for(auto i = _begin; i != _end; ++i) cin >> (*i); }
40
41 #endif
42
43 clock_t start_time;
44 string run_time()
45 { return to_string((clock() - (double)start_time) / CLOCKS_PER_SEC) + " sec"; }
46
47 #ifndef ONLINE_JUDGE
48 #include <sys/resource.h>
49 void resize_stack()
50 {
51     rlimit rlim;
52     getrlimit(RLIMIT_STACK, &rlim);
53     rlim.rlim_cur = (1 << 28);
54     setrlimit(RLIMIT_STACK, &rlim);
55 }
56 #else
57 void resize_stack() {};
58 #endif

```

## 8.7 Overloaded operators to accept 128 bit integer

```

1 typedef __uint128_t    ui128;
2 typedef __int128       i128;
3
4 template <class T> string to_string(T x) {
5     int sn = 1; if(x < 0) sn = -1, x *= sn; string s = "";
6     do { s = "0123456789"[x % 10] + s, x /= 10; } while(x);
7     return (sn == -1 ? "-" : "") + s;
8 }
9
10 auto str_to_int(string x) {
11     ui128 ret = (x[0] == '-' ? 0 : x[0] - '0');
12     for(int i = 1; i < x.size(); ++i) ret = ret * 10 + (x[i] - '0');
13     return (x[0] == '-' ? -1 * (i128)ret : ret);
14 }
15
16 istream & operator >> (istream & in, i128 & i) noexcept { string s; in >> s; i = str_to_int(s); return
    in; }
17 ostream & operator << (ostream & os, const i128 i) noexcept { os << to_string(i); return os; }
18 istream & operator >> (istream & in, ui128 & i) noexcept { string s; in >> s; i = str_to_int(s);
    return in; }
19 ostream & operator << (ostream & os, const ui128 i) noexcept { os << to_string(i); return os; }

```

## 8.8 Policy based data structures

```

1 #if __cplusplus >= 201402L
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 #endif
5
6 #if __cplusplus >= 201402L
7 using namespace __gnu_cxx;
8 using namespace __gnu_pbds;
9 #endif
10
11 template <class T, typename Comp = less <T> >
12 using indexed_set = tree <T, null_type, Comp, rb_tree_tag, tree_order_statistics_node_update>;

```

```

13
14 template <typename K, typename V, typename Comp = less <K>>
15 using indexed_map = tree <K, V, Comp, rb_tree_tag, tree_order_statistics_node_update>;

```

## 8.9 Pseudo random number generator

```

1 /** pseudo-random number generator | C++xx >= C++11 **/
2
3 mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
4
5 T myRand(T a, T b) {
6     return uniform_int_distribution <T> (a, b)(rng);
7 }

```

## 8.10 Stress test

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define endl '\n'
6
7 using i128 = __int128_t;
8 using i64 = int64_t;
9 using i32 = int32_t;
10 using i16 = int16_t;
11 using i8 = int8_t;
12
13 using u128 = __uint128_t;
14 using u64 = uint64_t;
15 using u32 = uint32_t;
16 using u16 = uint16_t;
17 using u8 = uint8_t;
18
19 void fast() {
20     ios_base::sync_with_stdio(false);
21     cin.tie(nullptr);
22 }
23
24 mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
25
26 /** 64-bit signed int Generator
27 **/
28 i64 int64(i64 a, i64 b) {
29     return uniform_int_distribution <i64> (a, b)(rng);
30 }
31
32 /** Customize your Generator depending on the input
33 **/
34 void gen () {
35     ofstream cout("input.in");
36     i32 t = 2;
37     cout << t << endl;
38
39     while (t--) {
40         i32 n = int64(1, 100), m = int64(1, 100);
41         cout << n << " " << m << endl;
42
43         while (m--) {
44             i32 u = int64(1, n), v = int64(1, n), c = int64(1, 4);
45             cout << u << " " << v << " " << c << endl;
46         }
47     }
48 }
49
50 i32 main (i32 arg, char* args[]) {
51     fast();
52
53     i32 tc = 0;
54     i32 limit = 100;
55     if(arg != 3) return 0;
56
57     string flags = "g++ -Wall -Wextra -Wshadow -Og -g -Ofast -std=c++17 -D_GLIBCXX_ASSERTIONS -DDEBUG -
    gdbg3 -fsanitize=address,undefined -fmax-errors=2 -o ";
58     string ex = ".cpp", bf, oz, pr;
59
60     bf = flags + args[1] + " " + args[1] + ex;
61     oz = flags + args[2] + " " + args[2] + ex;
62     char bff[bff.size() + 1];
63     char ozz[oz.size() + 1];
64     strcpy(bff, bf.c_str());
65     strcpy(oz, oz.c_str());

```

```

66
67 // compile command
68 system(bff);
69 system(ozz);
70
71 ex = ".out";
72 pr = "./";
73 bf = pr + args[1] + " < input.in > " + args[1] + ex;
74 oz = pr + args[2] + " < input.in > " + args[2] + ex;
75 strcpy(bff, bf.c_str());
76 strcpy(ozz, oz.c_str());
77
78 while (++tc <= limit) {
79     gen();
80     cerr << tc << endl;
81     // run command
82     system(bff);
83     system(ozz);
84

```

```

85 ifstream brute_forces("brute_force.out");
86 ifstream optimizes("optimized.out");
87
88 string brute_force, optimized;
89 getline(brute_forces, brute_force, (char)EOF);
90 getline(optimizes, optimized, (char)EOF);
91
92 if(brute_force != optimized) {
93     cerr << "Wrong Answer" << endl;
94     break;
95 } else if (tc == limit) {
96     cout << "Accepted insha'a Allah" << endl;
97 }
98 }
99 }

```

---