

# ICPC Team Reference Material

## Contents

<b>1</b>	<b>Setup</b>	<b>1</b>
1.1	Vimrc . . . . .	1
1.2	Capslock as Escape . . . . .	1
1.3	Compilation . . . . .	1
<b>2</b>	<b>Graph algorithms</b>	<b>2</b>
2.1	Adjacency list representation . . . . .	2
2.2	Depth first search (DFS) . . . . .	2
2.3	Breadth first search (BFS) . . . . .	2
2.4	0-1 BFS . . . . .	2
2.5	0-1 BFS (grid) . . . . .	2
2.6	Articulation points and bridges . . . . .	3
2.7	Bi-connected components . . . . .	3
2.8	Bipartite graph . . . . .	3
2.9	Bellman ford . . . . .	3
2.10	Connected components . . . . .	4
2.11	Cycle detection . . . . .	4
2.12	Directed cyclic graph into acyclic . . . . .	4
2.13	Dijkstra (dense graph) . . . . .	4
2.14	Dijkstra (grid) . . . . .	5
2.15	Dijkstra (negative weighted graph) . . . . .	5
2.16	Dijkstra (sparse graph) . . . . .	5
2.17	Edge classification . . . . .	5
2.18	Eulerian tour tree . . . . .	6
2.19	Floodfill . . . . .	6
2.20	Floyd warshall (all-pairs shortest path) . . . . .	6
2.21	Minimum spanning tree (Kruskal) . . . . .	6
2.22	Kth ancestor and lowest common ancestor (binary lifting) . . . . .	6
2.23	Lowest common ancestor (euler tour) . . . . .	7
2.24	Minimum vertex cover . . . . .	7
2.25	Shortest path faster algorithm (SPFA) . . . . .	7
2.26	Single source shortest path . . . . .	8
2.27	Single source shortest path (grid) . . . . .	8
2.28	Tarjan (strongly connected components) . . . . .	8
2.29	Tree diameter . . . . .	8
2.30	Tree diameter (weighted tree) . . . . .	8
2.31	Topological sort (kahns algorithm) . . . . .	9
2.32	Problems . . . . .	10
2.32.1	Restoring the path . . . . .	9
2.32.2	Tree distances . . . . .	9
2.32.3	Subtree sizes . . . . .	10
2.32.4	Shortest cycle . . . . .	10
2.32.5	Path sum for each node . . . . .	10
<b>3</b>	<b>Data structures</b>	<b>10</b>
3.1	Segment tree . . . . .	10
3.2	Merge sort tree . . . . .	11
3.3	Merge sort . . . . .	12
3.4	Sparse table . . . . .	12
3.5	Union find disjoint sets . . . . .	12
3.6	Bubble sort . . . . .	13
<b>4</b>	<b>Mathematics</b>	<b>13</b>
4.1	Pisano periodic sequence . . . . .	13
4.2	Euler totient function . . . . .	14
4.3	Extended wheel factorization . . . . .	14
4.4	Least prime factorization . . . . .	15
4.5	Mobius function . . . . .	15
4.6	Phi factorial . . . . .	15
4.7	Linear sieve . . . . .	15
4.8	Segmented sieve . . . . .	16
4.9	Miller Rabin test . . . . .	16
4.10	Stable marriage problem . . . . .	16

4.11	Euler totient sieve . . . . .	16
4.12	Mobius sieve . . . . .	17
4.13	Wheel sieve . . . . .	17

<b>5</b>	<b>String Processing</b>	<b>17</b>
5.1	Trie . . . . .	17
5.2	Knuth Morris Pratt (KMP) . . . . .	18

<b>6</b>	<b>Geometry</b>	<b>18</b>
6.1	Point . . . . .	18

<b>7</b>	<b>More advanced topics</b>	<b>19</b>
7.1	A*-Algorithm . . . . .	19
7.2	Mo's algorithm . . . . .	19
7.3	Square root decomposition . . . . .	20

<b>8</b>	<b>Miscellaneous</b>	<b>20</b>
8.1	Double comparison . . . . .	20
8.2	Fast input output . . . . .	20
8.3	Gcd & Lcm . . . . .	21
8.4	Modular calculations . . . . .	21
8.5	Overloaded operators to accept 128 Bit integer . . . . .	21
8.6	Policy based data structures . . . . .	21
8.7	stress test . . . . .	21

## 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 %:p:r<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"{$1,.cpp}
5 }
6 compile "$1"

```

## 2 Graph algorithms

### 2.1 Adjacency list representation

```

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

```

### 2.2 Depth first search (DFS)

```

1 void DFS(int node)
2 {
3     vis[node] = true;
4     for(int i = Head[node]; i; i = Next[i])
5         if(!vis[To[i]])
6             DFS(To[i]);
7 }

```

### 2.3 Breadth first search (BFS)

```

1 void BFS(int src)
2 {
3     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
4     memset(Par, -1, sizeof(Par[0]) * (n + 2));
5
6     queue<int> q;
7     q.push(src);
8     dis[src] = 0;
9
10    int u;
11    while(q.size())
12    {
13        u = q.front(); q.pop();
14        for(int i = Head[u]; i; i = Next[i]) if(dis[To[i]] == oo) {
15            dis[To[i]] = dis[u] + 1;
16            Par[To[i]] = u;
17            q.push(To[i]);
18        }
19    }
20 }

```

### 2.4 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 {
21     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
22     memset(Par, -1, sizeof(Par[0]) * (n + 2));
23     deque<int> Q;
24     Q.push_front(src);
25     dis[src] = 0;
26
27     int node;
28     while(Q.size())
29     {
30         node = Q.front(); Q.pop_front();
31         if(node == trg) return;
32
33         for(int i = Head[node]; i; i = Next[i])
34             if(dis[node] + Cost[i] < dis[To[i]])
35             {
36                 dis[To[i]] = dis[node] + Cost[i];
37                 if(Cost[i])
38                     Q.push_back(To[i]);
39                 else
40                     Q.push_front(To[i]);
41             }
42     }
43 }

```

### 2.5 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 {
16     memset(dis, 0x3f, sizeof(dis)); // memset(dis, 0x3f, n * m) we don't do that here
17     deque<pair<int, int>> Q;
18     dis[sr][sc] = 0;
19     Q.push_front({sr, sc});
20
21     int r, c, nr, nc, ncost;
22     while(Q.size())
23     {
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         {
29             nr = r + dr[i];
30             nc = c + dc[i];
31
32             if(!valid(nr, nc)) continue;
33             ncost = (i != grid[r][c]);
34
35             if(dis[r][c] + ncost < dis[nr][nc])
36             {
37                 dis[nr][nc] = dis[r][c] + ncost;
38
39                 if(ncost)
40                     Q.push_back({nr, nc});
41                 else

```

```

42     Q.push_front({nr, nc});
43     }
44     }
45     }
46     return oo;
47 }

```

## 2.6 Articulation points and bridges

```

1  int Head[N], Next[M], To[M], Cost[M], Par[N], dfs_num[N], dfs_low[N], ne, n, m, u, v, w, root,
    rootChildren, dfs_timer, bridgeInx;
2  bool Art[N];
3  vector < pair <int, int> > bridges(M);
4
5  void _clear() {
6      memset(Head, 0, sizeof(Head[0])) * (n + 2));
7      memset(dfs_num, 0, sizeof(dfs_num[0])) * (n + 2));
8      memset(Par, -1, sizeof(Par[0])) * (n + 2));
9      memset(Art, 0, sizeof(Art[0])) * (n + 2));
10     ne = dfs_timer = bridgeInx = 0;
11 }
12
13 void Tarjan(int node) {
14     dfs_num[node] = dfs_low[node] = ++dfs_timer;
15     for(int i = Head[node]; i; i = Next[i]) {
16         if(dfs_num[To[i]] == 0) {
17             if(node == root) ++rootChildren;
18             Par[To[i]] = node;
19             Tarjan(To[i]);
20             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
21
22             if(dfs_low[To[i]] >= dfs_num[node])
23                 Art[node] = true;
24
25             if(dfs_low[To[i]] > dfs_num[node])
26                 bridges[bridgeInx++] = make_pair(node, To[i]);
27         }
28         else if(To[i] != Par[node])
29             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
30     }
31 }
32
33 int main() {
34     for(int i = 1; i <= n; ++i)
35         if(dfs_num[i] == 0) {
36             root = i;
37             rootChildren = 0;
38             Tarjan(i);
39             Art[root] = (rootChildren > 1);
40         }
41
42     cout << "Art Points :\n";
43     for(int i = 1; i <= n; ++i)
44         if(Art[i])
45             cout << i << " ";
46
47     cout << "\nBridges :\n";
48     for(int i = 0; i < bridgeInx; ++i)
49         cout << bridges[i].first << " - " << bridges[i].second << endl;
50 }

```

## 2.7 Bi-connected components

```

1  int Head[N], Next[M], To[M], Par[N], dfs_num[N], dfs_low[N], ne, n, m, u, v, root, rootChildren,
    dfs_timer, Stack[N], top, ID;
2  bool Art[N];
3  vector < vector <int> > BiCCs(N), BiCCIDs(N);
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 _clear() {
12     memset(Head, 0, sizeof(Head[0])) * (n + 2);
13     memset(dfs_num, 0, sizeof(dfs_num[0])) * (n + 2);
14     memset(Par, -1, sizeof(Par[0])) * (n + 2);
15     memset(Art, 0, sizeof(Art[0])) * (n + 2);
16     ne = dfs_timer = top = ID = 0;
17     BiCCs = BiCCIDs = vector < vector <int> > (N);
18 }

```

```

19
20 void Tarjan(int node)
21 {
22     dfs_num[node] = dfs_low[node] = ++dfs_timer;
23     Stack[top++] = node;
24
25     for(int i = Head[node]; i; i = Next[i]) {
26         if(dfs_num[To[i]] == 0) {
27             if(node == root) ++rootChildren;
28             Par[To[i]] = node;
29             Tarjan(To[i]);
30
31             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
32             if(dfs_low[To[i]] >= dfs_num[node])
33             {
34                 Art[node] = true;
35                 ++ID;
36                 for(int x = -1; x ^ To[i];)
37                 {
38                     x = Stack[--top];
39                     BiCCIDs[x].emplace_back(ID);
40                     BiCCs[ID].emplace_back(x);
41                 }
42                 BiCCIDs[node].emplace_back(ID);
43                 BiCCs[ID].emplace_back(node);
44             }
45             else if(To[i] != Par[node])
46                 dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
47         }
48     }
49 }
50
51 int main()
52 {
53     for(int i = 1; i <= n; ++i)
54         if(dfs_num[i] == 0) {
55             root = i;
56             rootChildren = 0;
57             Tarjan(i);
58             Art[root] = (rootChildren > 1);
59         }
60
61     for(int i = 1; i <= ID; ++i) {
62         cout << "Component : " << i << " contains : ";
63         for(int j = 0; j < (int)BiCCs[i].size(); ++j)
64             cout << BiCCs[i][j] << " \n" << BiCCs[i].size() - 1;
65     }
66 }

```

## 2.8 Bipartite graph

```

1  bool checkBiPartite(int node, int par = 0) {
2      if(vis[node])
3          return color[par] != color[node];
4
5      color[node] = color[par] ^ 1;
6      vis[node] = true;
7      bool ok = true;
8      for(int i = Head[node]; i; i = Next[i])
9          if(To[i] != par)
10             ok &= checkBiPartite(To[i], node);
11     return ok;
12 }
13
14 int main() {
15     bool isBiPartite = true;
16     for(int i = 1; i <= n; ++i)
17         if(!vis[i])
18             isBiPartite &= checkBiPartite(i);
19     cout << (isBiPartite ? "YES" : "NO") << endl;
20 }

```

## 2.9 Bellman ford

```

1  // Bellman Ford Algorithm : In programming contests, the slowness of Bellman Fords and its negative
    cycle detection feature causes it to be used only to solve the SSSP problem on small graph
    which is not guaranteed to be free from negative weight cycle.
2
3  bool hasNC() {
4      for(int i = 1; i <= n; ++i)
5          for(int j = Head[i]; j; j = Next[j])
6              if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]])

```

```

7   return true;
8
9   return false;
10 }
11
12 bool Bellman_Ford(int src)
13 {
14     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
15     memset(Par, -1, sizeof(Par[0]) * (n + 2));
16
17     dis[src] = 0;
18     bool newRelaxation = true;
19
20     for(int i = 2; i <= n && newRelaxation; ++i) {
21         newRelaxation = false;
22         for(int i = 1; i <= n; ++i)
23             for(int j = Head[i]; j; j = Next[j])
24                 if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]]) {
25                     dis[To[j]] = dis[i] + Cost[j];
26                     Par[To[j]] = i;
27                     newRelaxation = true;
28                 }
29     }
30     return hasNC();
31 }

```

## 2.10 Connected components

```

1 void DFS(int node) {
2     visited[node] = true;
3     for(int e = Head[node]; e; e = Next[e])
4         if(!visited[To[e]])
5             DFS(To[e]);
6 }
7
8 int main() {
9     for(int node = 1; node <= n; ++node)
10         if(!visited[node])
11             ++CCs, DFS(node);
12     cout << CCs << endl;
13 }

```

## 2.11 Cycle detection

```

1 void DFS(int node, int parent = -1)
2 {
3     if(hasCycle != visited[node])
4         return;
5     visited[node] = true;
6
7     for(int i = Head[node]; i; i = Next[i])
8         if(To[i] != parent)
9             DFS(To[i], node);
10 }
11
12 int main() {
13     for(int i = 1; i <= n; ++i)
14         if(!visited[i])
15             DFS(i);
16     cout << (hasCycle ? "YES" : "NO") << endl;
17 }

```

## 2.12 Directed cyclic graph into acyclic

```

1 int HeadDAG[N], ToDAG[M], NextDAG[M], CostDAG[M], neDAG, Head[N], To[M], Next[M], Cost[M], dfs_num[N],
2   dfs_low[N], out[N], Stack[N], compID[N], compSize[N], ne, n, m, u, v, w, dfs_timer, top, ID;
3 bool in_stack[N];
4
5 void addEdge(int from, int to, int cost = 0) {
6     Next[++ne] = Head[from];
7     Head[from] = ne;
8     Cost[ne] = cost;
9     To[ne] = to;
10 }
11
12 void addEdgeDAG(int from, int to, int cost = 0) {
13     NextDAG[++neDAG] = HeadDAG[from];

```

```

13 HeadDAG[from] = neDAG;
14 CostDAG[ne] = cost;
15 ToDAG[neDAG] = to;
16 ++out[from];
17 }
18
19 void _clear() {
20     memset(Head, 0, sizeof(Head[0]) * (n + 2));
21     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
22     memset(compID, 0, sizeof(compID[0]) * (n + 2));
23     memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
24     memset(HeadDAG, 0, sizeof(HeadDAG[0]) * (n + 2));
25     memset(out, 0, sizeof(out[0]) * (n + 2));
26     ne = dfs_timer = top = neDAG = ID = 0;
27 }
28
29 void Tarjan(int node)
30 {
31     dfs_num[node] = dfs_low[node] = ++dfs_timer;
32     in_stack[Stack[top++] = node] = true;
33
34     for(int i = Head[node]; i; i = Next[i]) {
35         if(dfs_num[To[i]] == 0)
36             Tarjan(To[i]);
37
38         if(in_stack[To[i]])
39             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
40     }
41
42     if(dfs_num[node] == dfs_low[node]) {
43         ++ID;
44         for(int cur = -1; cur ^ node; ) {
45             in_stack[cur = Stack[--top]] = false;
46             compID[cur] = ID;
47             ++compSize[ID];
48         }
49     }
50 }
51
52 void Tarjan() {
53     for(int i = 1; i <= n; ++i)
54         if(dfs_num[i] == 0)
55             Tarjan(i);
56 }
57
58 void DFS(int node)
59 {
60     dfs_num[node] = 1;
61     for(int i = Head[node]; i; i = Next[i]) {
62         if(compID[node] != compID[To[i]])
63             addEdgeDAG(compID[node], compID[To[i]]);
64         if(dfs_num[To[i]] == 0)
65             DFS(To[i]);
66     }
67 }
68
69 void construct_dag() {
70     memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
71
72     for(int i = 1; i <= n; ++i)
73         if(dfs_num[i] == 0)
74             DFS(i);
75 }

```

## 2.13 Dijkstra (dense graph)

```

1 /** Dijkstra on dense graphs
2   * complexity : O(n^2 + m)
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 Dijkstra(int src, int V)
8 {
9     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
10    memset(Par, -1, sizeof(Par[0]) * (n + 2));
11
12    vector<bool> mark(V + 1, false);
13
14    dis[src] = 0;
15    for(int i = 1; i <= V; ++i) {
16        int node = 0;
17        for(int j = 1; j <= V; ++j)
18            if(!mark[j] && dis[j] < dis[node])
19                node = j;
20
21        if(dis[node] == INF) break;

```

```

22     mark[node] = true;
23     for(int i = Head[node]; i; i = Next[i])
24         if(dis[node] + Cost[i] < dis[To[i]])
25         {
26             dis[To[i]] = dis[node] + Cost[i];
27             Par[To[i]] = node;
28         }
29     }
30 }

```

## 2.14 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 {
15     memset(dis, 0x3f, sizeof(dis)); // memset(dis, 0x3f, n * m) we don't do that here
16
17     priority_queue<tuple<int, int, int>> Q;
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         {
28             nr = r + dr[i];
29             nc = c + dc[i];
30
31             if(!valid(nr, nc)) continue;
32
33             if(dis[r][c] + grid[nr][nc] < dis[nr][nc])
34             {
35                 dis[nr][nc] = dis[r][c] + grid[nr][nc];
36                 Q.push({-dis[nr][nc], nr, nc});
37             }
38         }
39     }
40 }
41

```

## 2.15 Dijkstra (negative weighted graph)

```

1  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
2  ll dis[N];
3
4  void Dijkstra(int src)
5  {
6      memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
7      memset(Par, -1, sizeof(Par[0]) * (n + 2));
8
9      priority_queue<pair<ll, int>> Q;
10     dis[src] = 0;
11     Q.push({-dis[src], src});
12
13     int node;
14     ll cost;
15     while(Q.size()) {
16         tie(cost, node) = Q.top(); Q.pop();
17         if((-cost) > dis[node]) continue;
18
19         for(int i = Head[node]; i; i = Next[i])
20             if(dis[node] + Cost[i] < dis[To[i]])
21             {
22                 dis[To[i]] = dis[node] + Cost[i];
23                 Q.push({-dis[To[i]], To[i]});
24                 Par[To[i]] = node;
25             }
26     }
27 }

```

## 2.16 Dijkstra (sparse graph)

```

1  /** Dijkstra on sparse graphs
2   - complexity : O(n + m)logn -> O(nlogn + 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  int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
8  ll dis[N];
9
10 void Dijkstra(int src, int trg)
11 {
12     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
13     memset(Par, -1, sizeof(Par[0]) * (n + 2));
14
15     priority_queue<pair<ll, int>> Q;
16     dis[src] = 0;
17     Q.push({-dis[src], src});
18
19     int node;
20     ll cost;
21     while(Q.size()) {
22         tie(cost, node) = Q.top(); Q.pop();
23
24         if((-cost) > dis[node]) continue; // lazy deletion
25         if(node == trg) return; // cheapest cost in case of positive weight edges
26
27         for(int i = Head[node]; i; i = Next[i])
28             if(dis[node] + Cost[i] < dis[To[i]])
29             {
30                 dis[To[i]] = dis[node] + Cost[i];
31                 Q.push({-dis[To[i]], To[i]});
32                 Par[To[i]] = node;
33             }
34     }
35 }

```

## 2.17 Edge classification

```

1  int Head[N], Next[M], To[M], Par[N], in_time[N], ne, n, m, u, v, dfs_timer;
2  char dfs_num[N];
3
4  void edgeClassification(int node)
5  {
6      dfs_num[node] = EXPLORED;
7      in_time[node] = ++dfs_timer;
8
9      for(int i = Head[node]; i; i = Next[i])
10     {
11         if(dfs_num[To[i]] == UNVISITED)
12         {
13             cout << "Tree Edge : " << node << " -> " << To[i] << endl;
14
15             Par[To[i]] = node;
16             edgeClassification(To[i]);
17         }
18         else if(dfs_num[To[i]] == VISITED)
19         {
20             /** Cross Edges only occur in directed graph */
21             if(in_time[To[i]] < in_time[node])
22                 cout << "Cross Edge : " << node << " -> " << To[i] << endl;
23             else
24                 cout << "Forward Edge : " << node << " -> " << To[i] << endl;
25         }
26         else if(dfs_num[To[i]] == EXPLORED)
27         {
28             if(Par[node] == To[i])
29                 cout << "Bi-Directional Edge : " << node << " -> " << To[i] << endl;
30             else
31                 cout << "Backward Edge : " << node << " -> " << To[i] << " (Cycle)" << endl;
32         }
33     }
34     dfs_num[node] = VISITED;
35 }
36
37 int main() {
38     for(int i = 1; i <= n; ++i) if(!dfs_num[i])
39         edgeClassification(i);
40 }
41

```

## 2.18 Eulerian tour tree

```

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

```

## 2.19 Floodfill

```

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])) return true;
23
24     return false;
25 }
26
27 int main() {
28     cout << (FloodFill(0, 0) ? "YES" : "NO") << endl;
29 }

```

## 2.20 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 int Par[N][N], n, m, u, v, tax;
11 ll adj[N][N], dis[N][N];
12
13 vector <int> restorePath(int st, int tr)
14 {
15     vector <int> path;
16     if(dis[st][tr] == INF) return path;
17
18     for(int i = tr; st ^ i; i = Par[st][i])
19         path.push_back(i);
20
21     path.push_back(st);
22     reverse(path.begin(), path.end());
23     return path;
24 }
25
26 void Floyd_Warshall()
27 {
28     for(int i = 1; i <= n; ++i)
29         for(int j = 1; j <= n; ++j)
30             Par[i][j] = i;
31
32     for(int k = 1; k <= n; ++k)
33         for(int i = 1; i <= n; ++i)
34             for(int j = 1; j <= n; ++j)
35                 if(dis[i][k] + dis[k][j] < dis[i][j])
36                 {
37                     dis[i][j] = dis[i][k] + dis[k][j];
38                     Par[i][j] = Par[k][j];
39                 }
40 }

```

## 2.21 Minimum spanning tree (Kruskal)

```

1 int n, m, u, v, w;
2 vector < tuple <int, int, int> > edges;
3 UnionFind uf;
4
5 pair < ll, vector < pair <int, int> > > Kruskal()
6 {
7     sort(edges.begin(), edges.end());
8
9     vector < pair <int, int> > mstEdges;
10    int from, to, cost;
11    ll minWiegth = 0;
12
13    for(tuple <int, int, int> edge : edges)
14    {
15        tie(cost, from, to) = edge;
16        if(uf.union_set(from, to))
17        {
18            minWiegth += cost;
19            mstEdges.push_back(make_pair(from, to));
20        }
21    }
22
23    if(mstEdges.size() == n - 1)
24        return make_pair(minWiegth, mstEdges);
25
26    return make_pair(-1, vector < pair <int, int> > ());
27 }

```

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

```

1 int Head[N], To[M], Next[M], Par[N], up[N][LOG + 1], Log[N], Level[N], ne, n, u, v, q;
2
3 void _clear() {
4     memset(Head, 0, sizeof(Head[0]) * (n + 2));
5     memset(Par, 0, sizeof(Par[0]) * (n + 2));
6     memset(Level, 0, sizeof(Level[0]) * (n + 2));
7     ne = 0;
8 }
9
10 int lastBit(int a) {
11     return (a & -a);
12 }
13
14 void logCalc()
15 {
16     Log[1] = 0;
17     for(int i = 2; i < N; ++i)
18         Log[i] = Log[i >> 1] + 1;
19 }
20
21 void DFS(int node, int depth = 0)
22 {
23     Level[node] = depth;
24     up[node][0] = Par[node]; // Par[root] = root
25
26     for(int i = 1; i <= LOG; ++i) {
27         up[node][i] = up[up[node][i - 1]][i - 1];
28     }
29
30     for(int i = Head[node]; i; i = Next[i]) if(To[i] != Par[node]) {
31         Par[To[i]] = node;
32         DFS(To[i], depth + 1);
33     }
34 }
35
36 int KthAncestor(int u, int k)
37 {
38     if(k > Level[u]) return -1;
39
40     for(int i = lastBit(k); k; k -= lastBit(k), i = lastBit(k))
41         u = up[u][Log[i]];
42
43     return u;
44 }
45
46 int LCA(int u, int v)
47 {
48     if(Level[u] < Level[v]) swap(u, v);
49     int k = Level[u] - Level[v];
50
51     u = KthAncestor(u, k);
52     if(u == v) return u;
53
54     for(int i = LOG; i >= 0; --i)
55         if(up[u][i] ^ up[v][i])
56             u = up[u][i];
57     u = up[u][i];
58     v = up[v][i];
59 }
60
61 return up[u][0];
62 }
63
64 int main()
65 {
66     logCalc();
67     for(int i = 1; i <= n; ++i) if(Par[i] == 0) {
68         Par[i] = i;
69         DFS(i);
70     }
71
72     cin >> q;
73     while(q--)
74     {
75         cin >> u >> v;
76         cout << LCA(u, v) << endl;
77     }
78 }

```

## 2.23 Lowest common ancestor (euler tour)

```

1 int Head[N], To[M], Next[M], Cost[M], ne, n, m, u, v, w, q;
2 int Last[N], First[N], euler_tour[N << 1], Height[N << 1], euler_timer;
3
4 void EulerianTour(int node, int depth = 0)
5 {
6     euler_tour[++euler_timer] = node;

```

```

7     Height[euler_timer] = depth;
8     First[node] = euler_timer;
9
10    for(int i = Head[node]; i; i = Next[i])
11        if(First[To[i]] == 0)
12        {
13            EulerianTour(To[i], depth + Cost[i]);
14
15            euler_tour[++euler_timer] = node;
16            Height[euler_timer] = depth;
17        }
18
19    Last[node] = euler_timer;
20 }
21
22 int main()
23 {
24     EulerianTour(1);
25     SparseTable <int> st(Height + 1, Height + euler_timer + 1, [&](int a, int b) { return a <= b; });
26
27     int l, r; cin >> q;
28     while(q--)
29     {
30         cin >> l >> r;
31
32         int left = Last[l];
33         int right = Last[r];
34         if(left > right) swap(left, right);
35
36         cout << euler_tour[ st.query(left, right) ] << endl;
37     }
38 }

```

## 2.24 Minimum vertex cover

```

1 bool DFS(int node, int par = -1) {
2
3     bool black = false;
4     for(int e = Head[node]; e; e = Next[e])
5         if(To[e] != par)
6             black |= DFS(To[e], node);
7
8     MVC += black;
9     return !black;
10 }

```

## 2.25 Shortest path faster algorithm (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 times.
11 */
12 int Head[N], Par[N], Next[M], To[M], Cost[M], Cnt[N], ne, n, m, u, v, st, tax;
13 ll dis[N];
14 bool Inq[N];
15
16 void _set() {
17     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
18     memset(Par, -1, sizeof(Par[0]) * (n + 2));
19     memset(Cnt, 0, sizeof(Cnt[0]) * (n + 2));
20     memset(Inq, 0, sizeof(Inq[0]) * (n + 2));
21 }
22
23 bool SPFA(int src)
24 {
25     _set();
26
27     deque <int> Q;
28     Q.push_front(src);
29
30     dis[src] = 0;
31     Cnt[src] = 1;
32     Inq[src] = 1;

```

```

32 int node;
33 while(Q.size()) {
34     node = Q.front(); Q.pop_front(); Inq[node] = 0;
35
36     for(int i = Head[node]; i; i = Next[i])
37         if(dis[node] + Cost[i] < dis[To[i]]) {
38             dis[To[i]] = dis[node] + Cost[i];
39             Par[To[i]] = node;
40
41             if(!Inq[To[i]])
42             {
43                 if(++Cnt[To[i]] == n)
44                     return true; // graph has a negative weight cycle
45
46                 if(Q.size() && dis[To[i]] > dis[Q.front()])
47                     Q.push_back(To[i]);
48                 else
49                     Q.push_front(To[i]);
50
51                 Inq[To[i]] = true;
52             }
53         }
54     }
55     return false;
56 }

```

## 2.26 Single source shortest path

```

1 int Head[N], Par[N], Next[M], To[M], ne, n, m, u, v, st, tr;
2 ll dis[N];
3
4 void BFS(int src)
5 {
6     memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
7     memset(Par, -1, sizeof(Par[0]) * (n + 2));
8
9     queue<int> Q;
10    Q.push(src);
11    dis[src] = 0;
12
13    int node;
14    while(Q.size()) {
15        node = Q.front(); Q.pop();
16        for(int i = Head[node]; i; i = Next[i])
17            if(dis[To[i]] == INF) {
18                dis[To[i]] = dis[node] + 1;
19                Par[To[i]] = node;
20                Q.push(To[i]);
21            }
22    }
23 }

```

## 2.27 Single source shortest path (grid)

```

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 int dis[N][N], n, m;
7 char Par[N][N];
8
9 bool valid(int r, int c) {
10     return r >= 1 && r <= n && c >= 1 && c <= m && dis[r][c] == oo;
11 }
12
13 void BFS(int sr, int sc)
14 {
15     memset(dis, 0x3f, sizeof(dis));
16     memset(Par, -1, sizeof(Par));
17
18     queue<pair<int, int>> Q;
19     dis[sr][sc] = 0;
20     Q.push({sr, sc});
21
22     int r, c, nr, nc;
23     while(Q.size()) {
24         tie(r, c) = Q.front(); Q.pop();
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         dis[nr][nc] = dis[r][c] + 1;
32         Par[nr][nc] = dir[i ^ 2];
33         Q.push({nr, nc});
34     }
35 }
36 }

```

## 2.28 Tarjan (strongly connected components)

```

1 int Head[N], To[M], Next[M], Cost[M];
2 int dfs_num[N], dfs_low[N];
3 int Stack[N], compID[N], compSize[N];
4 int ne, n, m, u, v, w;
5 int dfs_timer, top, ID;
6 bool in_stack[N];
7
8 void _clear() {
9     memset(Head, 0, sizeof(Head[0]) * (n + 2));
10    memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
11    memset(compID, 0, sizeof(compID[0]) * (n + 2));
12    memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
13    ne = dfs_timer = top = ID = 0;
14 }
15
16 void Tarjan(int node)
17 {
18     dfs_num[node] = dfs_low[node] = ++dfs_timer;
19     in_stack[Stack[top++] = node] = true;
20
21     for(int i = Head[node]; i; i = Next[i]) {
22         if(dfs_num[To[i]] == 0)
23             Tarjan(To[i]);
24
25         if(in_stack[To[i]])
26             dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
27     }
28
29     if(dfs_num[node] == dfs_low[node]) {
30         ++ID;
31         for(int cur = -1; cur ^ node; ) {
32             in_stack[cur = Stack[--top]] = false;
33             compID[cur] = ID;
34             ++compSize[ID];
35         }
36     }
37 }
38
39 void Tarjan() {
40     for(int i = 1; i <= n; ++i) if(dfs_num[i] == 0)
41         Tarjan(i);
42 }

```

## 2.29 Tree diameter

```

1 void DFS(int node, int cost, int par = -1) {
2     if(cost > diameter) diameter = cost, at = node;
3     for (int e = Head[node]; e; e = Next[e]) if(To[e] != par)
4         DFS(To[e], cost + 1, node);
5 }
6
7 int main() {
8     DFS(1, 0);
9     from = at, diameter = 0;
10    DFS(from, 0);
11    to = at;
12    cout << diameter << " " << from << " " << to << endl;
13 }

```

## 2.30 Tree diameter (weighted tree)

```

1 void DFS(int node, long long cost, int par = -1) {
2     if(cost > diameter) diameter = cost, at = node;
3     for (int e = Head[node]; e; e = Next[e])
4         if(To[e] != par)
5             DFS(To[e], cost + Cost[e], node);
6 }
7

```



```

8 int main() {
9     DFS(1, 011);
10    from = at, diameter = 0;
11    DFS(from, 011);
12    to = at;
13    cout << diameter << endl;
14 }

```

## 2.31 Topological sort (kahn's algorithm)

```

1 vector<int> kahn(int n)
2 {
3     vector<int> ready, ret;
4
5     for(int i = 1; i <= n; ++i)
6         if(!in[i])
7             ready.push_back(i);
8
9     int node;
10    while(!ready.empty())
11    {
12        node = ready.back(); ready.pop_back();
13        ret.push_back(node);
14
15        for(int i = Head[node]; i; i = Next[i])
16            if(--in[To[i]] == 0)
17                ready.push_back(To[i]);
18    }
19    return ret;
20 }
21
22 int main() {
23     vector<int> v = kahn(n);
24     if((int)v.size() == n)
25         for(int i : v)
26             cout << i << ' ';
27     else
28         cout << "not a DAG!" << endl;
29 }

```

## 2.32 Problems

### 2.32.1 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 }

```

### 2.32.2 Tree distances

```

1 int Head[N], Next[M], To[M], Par[N], ne, n, m, u, v, diameter, At, From;
2
3 int E[N << 1], H[N << 1], F[N], L[N], timer, SP[N << 1][LOG + 1], Log[N << 1];
4
5 void _clear() {
6     memset(Head, 0, sizeof(Head[0]) * (n + 2));
7     memset(Par, -1, sizeof(Par[0]) * (n + 2));
8     ne = 0;
9     timer = 0;
10 }
11
12 void EulerTour(int node, int depth = 0, int par = -1) {
13     E[++timer] = node;
14     H[timer] = depth;
15     F[node] = timer;
16
17     for(int i = Head[node]; i; i = Next[i])
18         if(To[i] != par) {
19             EulerTour(To[i], depth + 1, node);
20             E[++timer] = node;
21             H[timer] = depth;
22         }
23
24     L[node] = timer;
25 }
26
27 void dfs(int node, int depth = 0, int par = -1) {
28     if(depth > diameter) diameter = depth, At = node;
29     for(int i = Head[node]; i; i = Next[i])
30         if(To[i] != par)
31             dfs(To[i], depth + 1, node);
32 }
33
34 void bulid()
35 {
36     EulerTour(1);
37     dfs(1); From = At; diameter = 0; dfs(From);
38
39     Log[1] = 0;
40     for(int i = 2; i <= (n << 1); ++i)
41         Log[i] = Log[i >> 1] + 1;
42
43     for(int i = 1; i < (n << 1); ++i)
44         SP[i][0] = i;
45
46     int MaxLog = Log[(n << 1)];
47     for(int j = 1, k, h; j <= MaxLog; ++j) {
48         k = (1 << j);
49         h = (k >> 1);
50         for(int i = 1; i + k - 1 < (n << 1); ++i)
51             {
52                 const int & x = SP[i][j - 1];

```

```

53     const int & y = SP[i + h][j - 1];
54     SP[i][j] = H[x] <= H[y] ? x : y;
55     }
56 }
57 }
58 }
59
60 int query(int l, int r)
61 {
62     int d = r - l + 1;
63     int lg = Log[d];
64     int k = (1 << lg);
65
66     const int & x = SP[l][lg];
67     const int & y = SP[l + d - k][lg];
68
69     return (H[x] <= H[y] ? x : y);
70 }
71
72 int LCA(int u, int v) {
73     return query(u, v);
74 }
75
76 int distance(int u, int v) {
77     int l = F[u];
78     int r = F[v];
79     if(l > r) swap(l, r);
80
81     int ix = LCA(l, r);
82     return (H[l] + H[r] - H[ix] - H[ix]);
83 }
84
85 int main()
86 {
87     bulid();
88     for(int i = 1; i <= n; ++i)
89         cout << max(distance(i, At), distance(i, From)) << " \n"[i == n];
90 }

```

### 2.32.3 Subtree sizes

```

1  int Head[N], Next[M], To[M], Par[N], sbtree_size[N], ne, n, m, u, v, w;
2
3  void dfs(int node, int par = -1) {
4      sbtree_size[node] = 1;
5      for(int i = Head[node]; i; i = Next[i])
6          if(To[i] != par) {
7              dfs(To[i], node);
8              sbtree_size[node] += sbtree_size[To[i]];
9          }
10 }
11
12 int main()
13 {
14     dfs(1);
15     for(int i = 1; i <= n; ++i)
16         cout << sbtree_size[i] - 1 << " \n"[i == n];
17 }

```

### 2.32.4 Shortest cycle

```

1  /** for each node run BFS and minimize the cycle length
2  **/
3
4  int BFS(int src)
5  {
6      memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
7      memset(Par, -1, sizeof(Par[0]) * (n + 2));
8
9      queue<int> Q;
10     Q.push(src);
11     dis[src] = 0;
12
13     int node, ret = oo;
14     while(Q.size())
15     {
16         node = Q.front(); Q.pop();
17         for(int i = Head[node]; i; i = Next[i])
18         {
19             if(dis[To[i]] != oo) {
20                 if(Par[node] != To[i]) {
21                     if(dis[node] + 1 + dis[To[i]] < ret)
22                         ret = dis[node] + 1 + dis[To[i]];

```

```

23         }
24         continue;
25     }
26
27     dis[To[i]] = dis[node] + 1;
28     Par[To[i]] = node;
29     Q.push(To[i]);
30 }
31 }
32 return ret;
33 }

```

### 2.32.5 Path sum for each node

```

1  int Head[N], Next[M], To[M], ne, u, v, n, m, subtree_size[N], level[N];
2  ll dis[N];
3
4  void dfs(int node, int par = -1) {
5      subtree_size[node] = 1;
6      for(int i = Head[node]; i; i = Next[i]) if(To[i] != par) {
7          level[To[i]] = level[node] + 1;
8          dfs(To[i], node);
9          subtree_size[node] += subtree_size[To[i]];
10     }
11 }
12
13 void reRoot(int node, ll pd, int par = -1) {
14     dis[node] = pd;
15     for(int i = Head[node]; i; i = Next[i]) if(To[i] != par) {
16         reRoot(To[i], pd - subtree_size[To[i]] + (n - subtree_size[To[i]]), node);
17     }
18 }
19
20 void get_dis()
21 {
22     dfs(1);
23     ll pd = 0;
24     for(int i = 1; i <= n; ++i)
25         pd += level[i];
26
27     reRoot(1, pd);
28     for(int i = 1; i <= n; ++i)
29         cout << dis[i] << " \n"[i == n];
30 }

```

## 3 Data structures

### 3.1 Segment tree

```

1  template <class T, class F = function <T(const T &, const T &)>>
2  class SegmentTree
3  {
4      vector<T> _A;
5      vector<T> ST;
6      vector<T> LT;
7      F func;
8      int _N;
9
10 public :
11     template <class iter>
12     SegmentTree(iter _begin, iter _end, const F _func = [](T a, T b) {return a <= b ? a : b;}) : func(
13         _func)
14     {
15         _N = distance(_begin, _end);
16         _N = (1 << (int)ceil(log2(_N)));
17
18         _A.assign(_N + 1, 0);
19         ST.assign(_N << 1, 0);
20         LT.assign(_N << 1, 0);
21
22         _typeof(_begin) i = _begin;
23         for(int j = 1; i != _end; ++i, ++j)
24             _A[j] = *i;
25
26         build(1, 1, _N);
27
28     void build(int p, int l, int r)
29     {

```

## 3.2 Merge sort tree

```

30     if(l == r) {
31         ST[p] = _A[l];
32         return;
33     }
34
35     int mid = (l + r) >> 1;
36
37     build(p + p, l, mid);
38     build(p + p + 1, mid + 1, r);
39
40     const T & x = ST[p + p];
41     const T & y = ST[p + p + 1];
42
43     ST[p] = func(x, y);
44 }
45
46 void update_range(int ul, int ur, int delta) {
47     update_range(ul, ur, delta, 1, 1, _N);
48 }
49
50 T query(int ql, int qr) {
51     return query(ql, qr, 1, 1, _N);
52 }
53
54 void update_point(int inx, int delta)
55 {
56     inx += _N - 1;
57     ST[inx] = delta;
58
59     while(inx > 1) {
60         inx >>= 1;
61
62         const T & x = ST[inx + inx];
63         const T & y = ST[inx + inx + 1];
64
65         ST[inx] = func(x, y);
66     }
67 }
68
69 private :
70 void update_range(int ul, int ur, int delta, int p, int l, int r)
71 {
72     if(r < ul || ur < l)
73         return;
74
75     if(ul <= l && r <= ur) {
76         ST[p] += delta;
77         LT[p] += delta;
78         return;
79     }
80
81     propagate(p);
82
83     int mid = (l + r) >> 1;
84
85     update_range(ul, ur, delta, p + p, l, mid);
86     update_range(ul, ur, delta, p + p + 1, mid + 1, r);
87
88     const T & x = ST[p + p];
89     const T & y = ST[p + p + 1];
90
91     ST[p] = func(x, y);
92 }
93
94 T query(int ql, int qr, int p, int l, int r)
95 {
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 };

```

```

1  class SegmentTree
2  {
3      vector <vector <int>> > sTree;
4      vector <int> localArr;
5      int NP2, oo = 0x3f3f3f3f;
6
7  public :
8      template <class T>
9      SegmentTree(T _begin, T _end)
10     {
11         NP2 = 1;
12         int n = _end - _begin;
13         while(NP2 < n) NP2 <=<= 1;
14
15         sTree.assign(NP2 << 1, vector <int> ());
16         localArr.assign(NP2 + 1, 0);
17
18         _typeof(_begin) i = _begin;
19         for(int j = 1; i != _end; i++, ++j)
20             localArr[j] = *i;
21
22         build(1, 1, NP2);
23     }
24
25     void build(int p, int l, int r)
26     {
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     {
45         if(isOutside(ql, qr, l, r))
46             return 0;
47
48         if(isInside(ql, qr, l, r)) {
49             return sTree[p].end() - upper_bound(sTree[p].begin(), sTree[p].end(), k);
50         }
51
52         return query(ql, qr, k, left(p), l, mid(l, r)) +
53             query(ql, qr, k, right(p), mid(l, r) + 1, r);
54     }
55
56     void merge(int p)
57     {
58         vector <int> & L = sTree[left(p)];
59         vector <int> & R = sTree[right(p)];
60
61         int l_size = L.size();
62         int r_size = R.size();
63         int p_size = l_size + r_size;
64
65         L.push_back(INT_MAX);
66         R.push_back(INT_MAX);
67
68         sTree[p].resize(p_size);
69
70         for(int k = 0, i = 0, j = 0; k < p_size; ++k)
71             if(L[i] <= R[j])
72                 sTree[p][k] = L[i], i += (L[i] != INT_MAX);
73             else
74                 sTree[p][k] = R[j], j += (R[j] != INT_MAX);
75
76         L.pop_back();
77         R.pop_back();
78     }
79
80     inline bool isInside(int ql, int qr, int sl, int sr) {
81         return (ql <= sl && sr <= qr);
82     }
83
84     inline bool isOutside(int ql, int qr, int sl, int sr) {
85         return (sr < ql || qr < sl);
86     }

```

```

87
88     inline int mid (int l, int r) {
89         return ((l + r) >> 1);
90     }
91
92     inline int left(int p) {
93         return (p << 1);
94     }
95
96     inline int right(int p) {
97         return ((p << 1) | 1);
98     }
99 };

```

### 3.3 Merge sort

```

1  ll inversions;
2
3  template <class T>
4  void merge(T localArr [], int l, int mid, int r)
5  {
6      int l_size = mid - l + 1;
7      int r_size = r - mid;
8
9      T L[l_size + 1];
10     T R[r_size + 1];
11
12     for(int i = 0; i < l_size; ++i) L[i] = localArr[i + l];
13     for(int i = 0; i < r_size; ++i) R[i] = localArr[i + mid + 1];
14
15     T Mx;
16     if(sizeof(T) == 4) Mx = INT_MAX;
17     else Mx = LONG_MAX;
18
19     L[l_size] = R[r_size] = Mx;
20
21     for(int k = 1, i = 0, j = 0; k <= r; ++k)
22         if(L[i] <= R[j])
23             localArr[k] = L[i], i += (L[i] != Mx);
24         else
25             localArr[k] = R[j], j += (R[j] != Mx), inversions += l_size - i;
26 }
27
28 template <class T>
29 void merge_sort(T localArr [], int l, int r)
30 {
31     if(r - l)
32     {
33         int mid = (l + r) >> 1;
34         merge_sort(localArr, l, mid);
35         merge_sort(localArr, mid + 1, r);
36         merge(localArr, l, mid, r);
37     }
38 }
39
40 template <class T>
41 void merge_sort(T _begin, T _end)
42 {
43     const int sz = _end - _begin;
44     __typeof(*_begin) localArray[sz];
45
46     __typeof(_begin) k = _begin;
47     for(int i = 0; k != _end; ++i, ++k)
48         localArray[i] = *k;
49
50     merge_sort(localArray, 0, sz - 1);
51
52     k = _begin;
53     for(int i = 0; k != _end; ++i, ++k)
54         *k = localArray[i];
55 }

```

### 3.4 Sparse table

```

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

```

```

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

```

### 3.5 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
8   find_set(u) returns the unique representative of the partition containing u.
9   same_set(u, v) returns whether elements u and v belong to the same partition.
10  union_set(u, v) replaces the partitions containing u and v with a single new
11  partition consisting of the union of elements in the original partitions.
12
13  Time Complexity:
14  O(a(n)) per call to find_set(), same_set(), and union_set(), where n is the
15  number of elements, and a(n) is the extremely slow growing inverse of the Ackermann function
16  (effectively a very small constant for all practical values of n).
17
18  Space Complexity:
19  O(n) for storage of the disjoint set forest elements.
20  O(1) auxiliary for all operations.
21  **/
22
23  class UnionFind
24  {
25      vector <int> par;
26      vector <int> siz;
27      int num_sets;
28      size_t sz;

```

```

29 public:
30 UnionFind() : par(1, -1), siz(1, 1), num_sets(0), sz(0) {}
31 UnionFind(int n) : par(n + 1, -1), siz(n + 1, 1), num_sets(n), sz(n) {}
32
33 int find_set(int u)
34 {
35     assert(u <= sz);
36
37     int leader;
38     for(leader = u; ~par[leader]; leader = par[leader]);
39
40     for(int next = par[u]; u != leader; next = par[next]) {
41         par[u] = leader;
42         u = next;
43     }
44     return leader;
45 }
46
47 bool same_set(int u, int v) {
48     return find_set(u) == find_set(v);
49 }
50
51 bool union_set(int u, int v) {
52     if(same_set(u, v)) return false;
53
54     int x = find_set(u);
55     int y = find_set(v);
56
57     if(siz[x] < siz[y]) swap(x, y);
58
59     par[y] = x;
60     siz[x] += siz[y];
61
62     --num_sets;
63     return true;
64 }
65
66 int number_of_sets() {
67     return num_sets;
68 }
69
70 int size_of_set(int u) {
71     return siz[find_set(u)];
72 }
73
74 size_t size() {
75     return sz;
76 }
77
78 void clear() {
79     par.clear();
80     siz.clear();
81     sz = num_sets = 0;
82 }
83
84 void assign(size_t n) {
85     par.assign(n + 1, -1);
86     siz.assign(n + 1, 1);
87     sz = num_sets = n;
88 }
89
90 map< int, vector<int> > groups(int st) {
91     map< int, vector<int> > ret;
92
93     for(size_t i = st; i < sz + st; ++i)
94         ret[find_set(i)].push_back(i);
95
96     return ret;
97 }
98 };

```

## 3.6 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
13     __typeof(_begin) k = _begin;
14     for(int j = 0; k != _end; ++k, ++j)

```

```

15     localArray[j] = *k;
16
17     round = min(round, sz);
18     for(int i = 0; i < round; ++i) /* n rounds -> n_th element */
19         for(int j = 0; j < sz - 1; ++j) if(localArray[j] > localArray[j + 1])
20             swap(localArray[j], localArray[j + 1]);
21
22     k = _begin;
23     for(int j = 0; k != _end; ++k, ++j)
24         *k = localArray[j];
25 }
26
27 /**
28  * After the first round of the algorithm, the largest element will be in the correct
29  * position, and in general, after k rounds, the k largest elements will be in the
30  * correct positions.
31  */

```

## 4 Mathematics

### 4.1 Pisano periodic sequence

```

1 template<class T>
2 using matrix = vector< vector<T> >;
3
4 template<class T> string to_string(T x) {
5     int sn = 1;
6     if(x < 0) sn = -1, x *= sn;
7     string s = "";
8     do {
9         s = "0123456789"[x % 10] + s, x /= 10;
10    } while(x);
11    return (sn == -1 ? "-" : "") + s;
12 }
13
14 auto str_to_int(string x) {
15     u128 ret = (x[0] == '-' ? 0 : x[0] - '0');
16     for(int i = 1; i < (int)x.size(); ++i) ret = ret * 10 + (x[i] - '0');
17     return (x[0] == '-' ? -1 * (i128)ret : ret);
18 }
19
20 istream & operator >> (istream & in, i128 & i) noexcept {
21     string s;
22     in >> s;
23     i = str_to_int(s);
24     return in;
25 }
26
27 ostream & operator << (ostream & os, const i128 i) noexcept {
28     os << to_string(i);
29     return os;
30 }
31
32 void Fast() {
33     cin.sync_with_stdio(0);
34     cin.tie(0);
35     cout.tie(0);
36 }
37
38 ll n;
39 vector<int> primes;
40 matrix<ll> fibMatrix = {{1, 1},
41     {1, 0}};
42 };
43
44 i128 gcd(i128 a, i128 b) {
45     while (a && b)
46         a > b ? a %= b : b %= a;
47     return a + b;
48 }
49
50 i128 lcm(i128 a, i128 b) {
51     return a / gcd(a, b) * b;
52 }
53
54 vector< array<ll, 2> > factorize(ll x) {
55     vector< array<ll, 2> > ret;
56     for(int i = 0; ll * primes[i] * primes[i] <= x; ++i) {
57         if(x % primes[i]) continue;
58
59         int cnt = 0;
60         while (x % primes[i] == 0) {
61             cnt++;
62             x /= primes[i];

```

```

63     }
64     ret.push_back({primes[i], cnt});
65 }
66
67 if(x > 1) ret.push_back({x, 1});
68 return ret;
69 }
70
71 matrix <ll> MatMul(matrix <ll> A, matrix <ll> B, ll mod) {
72     int ra = A.size(), cb = B[0].size(), ca = A[0].size();
73     matrix <ll> C(ra, vector <ll> (cb));
74
75     for(int i = 0; i < ra; ++i)
76         for (int j = 0; j < cb; ++j) {
77             C[i][j] = 0;
78             for(int k = 0; k < ca; ++k)
79                 C[i][j] = (C[i][j] + (ll28)A[i][k] * B[k][j]);
80         }
81
82     matrix <ll> ret(ra, vector <ll> (cb));
83     for(int i = 0; i < ra; ++i)
84         for (int j = 0; j < cb; ++j)
85             ret[i][j] = C[i][j] % mod;
86
87     return ret;
88 }
89
90 matrix <ll> MatPow(matrix <ll> A, ll p, ll mod) {
91     int r = A.size(), c = A[0].size();
92     assert(r == c && p);
93     matrix <ll> result = A;
94     p--;
95
96     while(p) {
97         if(p & 1ll) result = MatMul(result, A, mod);
98         A = MatMul(A, A, mod);
99         p >>= 1ll;
100     }
101     return result;
102 }
103
104 ll28 ModExp(ll28 a, ll p) {
105     ll28 result = 1;
106     while(p) {
107         if(p & 1ll) result = result * a;
108         a *= a;
109         p >>= 1ll;
110     }
111     return result;
112 }
113
114 ll nthFib(ll n, ll mod) {
115     return MatPow(fibMatrix, n, mod)[0][1];
116 }
117
118 bool is_period(ll n, ll mod) {
119     return nthFib(n, mod) == 0 && nthFib(n + 1, mod) == 1;
120 }
121
122 ll solver(ll x, ll mod) {
123     vector < array <ll, 2> > factors = factorize(x);
124     for(int i = 0; i < (int)factors.size(); ++i) {
125         while(x % factors[i][0] == 0 && is_period(x / factors[i][0], mod))
126             x /= factors[i][0];
127     }
128     return x;
129 }
130
131 ll pisano_prime(ll val) {
132     if(val == 2) return 3;
133     if(val == 5) return 20;
134     if(val % 10 == 1 || val % 10 == 9)
135         return solver(val - 1, val);
136
137     return solver(2 * (val + 1), val);
138 }
139
140 const int N = 1e7 + 9;
141 bitset <N> isPrime;
142
143 void Precomputation_Sieve() {
144     isPrime.set();
145     int _sqrt = sqrtl(N);
146
147     for(int i = 5; i <= _sqrt; i += 6) {
148         if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
149         i += 2;
150         if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
151         i -= 2;
152     }
153 }
154

```

```

155 vector <int> Primes(int n) {
156     vector <int> _Primes;
157
158     if(n >= 2) _Primes.push_back(2);
159     if(n >= 3) _Primes.push_back(3);
160
161     for (int i = 5; i <= n; i += 6) {
162         if(isPrime[i]) _Primes.push_back(i);
163         i += 2;
164         if(isPrime[i]) _Primes.push_back(i);
165         i -= 2;
166     }
167     return _Primes;
168 }
169
170 void initialize()
171 {
172     Precomputation_Sieve();
173     primes = Primes(N);
174 }
175
176 void Solve() {
177     initialize();
178
179     cin >> n;
180     vector < array <ll, 2> > factors = factorize(n);
181
182     ll28 ans = 1;
183     for (int i = 0; i < (int)factors.size(); ++i) {
184         ans = lcm(ans, (ll28)pisano_prime(factors[i][0]) * ModExp(factors[i][0], factors[i][1] - 1));
185     }
186     cout << ans << endl;
187 }

```

## 4.2 Euler totient function

```

1 int lp[N], Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
2
3 void linear_sieve(int n) {
4     for (int i = 2; i <= n; ++i) {
5         if (lp[i] == 0) {
6             lp[i] = Primes[pnx++] = i;
7         }
8         for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
9             lp[comp] = Primes[j];
10        }
11    }
12 }
13
14 ll Phi(ll a) { // for Queries
15     ll ret = a, p;
16     for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
17         if (p * p > a) break;
18         if (a % p) continue;
19         ret -= ret / p;
20         while (a % p == 0) a /= p;
21     }
22     if (a > 1) ret -= ret / a;
23     return ret;
24 }

```

## 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 Mobius function

```

1  /**
2   Constraints:
3   1 <= x <= 1e7
4   2 <= n <= 1014
5
6   Time Complexity:
7   linear_sieve takes O(n)
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     }

```

```

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
28 int mobius(ll n) {
29     ll p, pp;
30     char mob = 1;
31     for (int i = 0; i < pnx && (p = Primes[i], pp = p * p, true); ++i) {
32         if (pp > n) break;
33         if (n % p) continue;
34         if (n % pp == 0) return 0;
35         n /= p;
36         mob = -mob;
37     }
38     if (n > 1) mob = -mob;
39     return mob;
40 }

```

## 4.6 Phi factorial

```

1  /**
2   Constraints:
3   1 <= x <= 1e7
4   2 <= n <= 1e7
5
6   Time Complexity:
7   linear_sieve takes O(n)
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         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
28 ll phi_factorial(int n) {
29     ll ret = 1;
30     for (int i = 2; i <= n; ++i) {
31         ret = ret * (lp[i] == 1 ? i - 1 : i);
32     }
33     return ret;
34 }

```

## 4.7 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 }

```

## 4.8 Segmented sieve

```

1 int lp[N];
2 int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
3 bool isPrime[N];
4
5 void linear_sieve(int n) {
6     for (int i = 2; i <= n; ++i) {
7         if (lp[i] == 0) {
8             lp[i] = Primes[pnx++] = i;
9         }
10        for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
11            lp[comp] = Primes[j];
12        }
13    }
14 }
15
16 vector<ll> segmented_sieve(ll l, ll r) {
17     l += l == 1;
18     int limit = r - l + 1;
19     vector<ll> ret;
20     memset(isPrime, true, sizeof(isPrime));
21
22     ll p;
23     for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
24         for (ll j = max(p * p, (l + p - 1) / p * p); j <= r; j += p)
25             isPrime[j - l] = false;
26     }
27
28     for (int i = 0; i < limit; ++i)
29         if (isPrime[i])
30             ret.emplace_back(i + l);
31     return ret;
32 }

```

## 4.9 Miller Rabin test

```

1 ll ModExp(ll base, ll e, ll mod)
2 {
3     ll result;
4     base %= mod;
5     for(result = 1; e; e >= 1ll)
6     {
7         if(e & 1ll)
8             result = ((ll28)result * base) % mod;
9         base = ((ll28)base * base) % mod;
10    }
11    return result;
12 }
13
14 bool CheckComposite(ll n, ll p, ll d, int r)
15 {
16     ll a = ModExp(p, d, n);
17     if(a == 1 || a == n - 1)
18         return false;
19     for(int i = 1; i < r; ++i)
20     {
21         a = ((ll28)a * a) % n;
22         if(a == n - 1)
23             return false;
24     }
25     return true;
26 }
27
28 bool Miller(ll n)
29 {
30     if(n < 2) return false;
31
32     int r; ll d;
33     for(r = 0, d = n - 1; (d & 1ll) == 0; d >= 1ll, ++r);
34
35     for(int p : {2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 37})
36     {
37         if(n == p)
38             return true;
39         if(CheckComposite(n, p, d, r))
40             return false;
41     }
42     return true;
43 }
44
45 int main()
46 {

```

```

47     cout << (Miller(n) ? "Yes, it is Prime" : "No, it is not a prime") << endl;
48 }

```

## 4.10 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 {
23     _clear();
24
25     cin >> n;
26     for(int i = 1; i <= n; ++i)
27         for(int j = 1; j <= n; ++j)
28             cin >> pref[i][j];
29
30     for(int i = 1; i <= n; ++i)
31         for(int j = 1; j <= n; ++j) {
32             cin >> v;
33             order[i][v] = j;
34         }
35
36     for(int i = 1; i <= n; ++i)
37         Q.push(i);
38
39     int man, woman;
40     while(Q.size())
41     {
42         woman = Q.front(); Q.pop();
43         man = pref[woman][++Next[woman]];
44
45         if(!wife[man] || order[man][woman] < order[man][wife[man]])
46             engage(man, woman);
47         else
48             Q.push(woman);
49     }
50
51     for(int i = 1; i <= n; ++i)
52         cout << husband[i] << endl;
53 }
54
55 /**
56 1
57 3
58 3 2 1
59 2 1 3
60 2 1 3
61 3 1 2
62 1 2 3
63 2 3 1
64 -----
65 3
66 2
67 1
68 **/

```

## 4.11 Euler totient 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 }

```

## 4.12 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 }

```

## 4.13 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             if (c == sz - 1) c = 0;
67         }
68     }
69 }

```

## 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 public:
11     Trie(char lett = '\0') {
12         memset(children, 0, sizeof(children));
13         prefixes = words = 0;
14         iseow = false;
15         cur_letter = lett;
16     }
17
18     void insert(string &str) { // O(l)
19         Trie* cur = this;
20         int inx, strsz = str.size();
21         for (int i = 0; i < strsz; ++i) {
22             inx = str[i] - 'a';
23             if (cur->children[inx] == nullptr)
24                 cur->children[inx] = new Trie(str[i]);
25
26             cur = cur->children[inx];
27             cur->prefixs++;
28         }
29         cur->iseow = true;
30         cur->words++;
31     }
32
33     int search_word(string &str) { // O(l)
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->prefixes;
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]->prefixes == 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
    end of the given prefix
92     if(node->iseow) {
93         if(occurrence.size() < 10) {
94             occurrence.push(make_pair(node->words, s));
95         } else {
96             if(node->words > occurrence.top().first) {
97                 occurrence.pop();
98                 occurrence.push(make_pair(node->words, s));
99             }
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  class point
2  {
3  public :
4      ld x, y;
5
6      point() = default;
7      point(ld _x, ld _y) : x(_x), y(_y) {}
8
9      bool operator < (point other) const {
10         if(fabs(x - other.x) > EPS) // if(x != other.x)
11             return x < other.x;
12         return y < other.y;
13     }
14 }

```

```

15 bool operator == (point other) const {
16     return ((fabs(x - other.x) < EPS) && (fabs(y - other.y) < EPS)); // " < EPS " equal to " == zero "
17 }
18
19 bool operator > (point other) const {
20     if(fabs(x - other.x) > EPS)
21         return x > other.x;
22     return y > other.y;
23 }
24
25 ld dist(point other) { // Euclidean distance
26     ld dx = this->x - other.x;
27     ld dy = this->y - other.y;
28     return sqrtl(dx * dx + dy * dy);
29 }
30
31 ld DEG_to_RAD(ld theta) {
32     return theta * PI / 180.0;
33 }
34
35 ld RAD_to_DEG(ld theta) {
36     return theta * 180.0 / PI;
37 }
38
39 point rotate(ld theta) {
40     ld rad = DEG_to_RAD(theta);
41     return point(cos(theta) * x - sin(theta) * y,
42                 sin(theta) * x + cos(theta) * y);
43 }
44 };

```

## 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];
10 int dis[N][N], n, m, si, sj, ti, tj;
11 char Par[N][N];
12
13 vector< pair<int, int> > restorePath(int sr, int sc, int tr, int tc)
14 {
15     vector< pair<int, int> > ret;
16     if(dis[tr][tc] == oo) return ret;
17
18     for(char i = Par[tr][tc]; (sr ^ tr) || (sc ^ tc); i = Par[tr][tc])
19     {
20         ret.push_back({tr, tc});
21         tr += dr[inv[i]];
22         tc += dc[inv[i]];
23     }
24
25     ret.push_back({sr, sc});
26     reverse(ret.begin(), ret.end());
27     return ret;
28 }
29
30 bool valid(int r, int c) {
31     return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] != '%';
32 }
33
34 /** admissible heuristic **/
35 int manhattanDistance(int x1, int y1, int x2, int y2) {
36     return (abs(x1 - x2) + abs(y1 - y2));
37 }
38
39 int Astar(int sr, int sc, int tr, int tc)
40 {
41     memset(dis, 0x3f, sizeof(dis));
42     memset(Par, -1, sizeof(Par));
43
44     priority_queue<tuple<int, int, int> > Q;
45
46     dis[sr][sc] = 0;
47     Q.push({-manhattanDistance(sr, sc, tr, tc), sr, sc});
48
49     int hcost, r, c, nr, nc;

```

```

50 while(Q.size())
51 {
52     tie(hcost, r, c) = Q.top(); Q.pop();
53     if(r == tr && c == tc) return dis[r][c];
54
55     for(int i = 0; i < 4; ++i)
56     {
57         nr = r + dr[i];
58         nc = c + dc[i];
59
60         if(!valid(nr, nc)) continue;
61
62         if(dis[r][c] + 1 < dis[nr][nc])
63         {
64             dis[nr][nc] = dis[r][c] + 1;
65             Par[nr][nc] = dir[i ^ 2];
66             Q.push({-dis[nr][nc] -manhattanDistance(nr, nc, tr, tc), nr, nc});
67         }
68     }
69
70     return -1;
71 }
72
73 void Solve()
74 {
75     cin >> si >> sj >> ti >> tj >> n >> m;
76
77     for(int i = 0; i < n; ++i)
78         for(int j = 0; j < m; ++j)
79             cin >> grid[i][j];
80
81     cout << Astar(si, sj, ti, tj) << endl;
82     vector< pair<int, int> > path = restorePath(si, sj, ti, tj);
83
84     for(auto point : path)
85         cout << point.first << " " << point.second << endl;
86 }
87
88 /**
89  P -> strat
90  . -> target
91
92  input:
93  0 2 2 3 5 5
94  %%P%-
95  $---
96  $---
97  %%%%
98  -----
99
100 output:
101  3
102  0 2
103  1 2
104  2 2
105  2 3
106
107 **/

```

### 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  {
7      int l, r, id, blk;
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
30 void remove(int id) {
31     cur -= (--freq[id] == 0);
32 }
33
34 int get_res() {
35     return cur;
36 }
37
38 int cur_l, cur_r, l, r, n, q, a[N];
39
40 void Solve()
41 {
42     cin >> n;
43     for(int i = 1; i <= n; ++i) cin >> a[i];
44
45     cin >> q;
46     for(int i = 1; i <= q; ++i) {
47         cin >> l >> r;
48         queries[i] = query(l, r, i);
49     }
50
51     sort(queries + 1, queries + 1 + q);
52
53     cur_l = 1, cur_r = 0; // assign to right invalid index
54     for(int i = 1; i <= q; ++i)
55     {
56         int ql = queries[i].l;
57         int qr = queries[i].r;
58
59         // Add right
60         while(cur_r < qr) add(a[++cur_r]);
61         // Add left
62         while(cur_l > ql) add(a[--cur_l]);
63         // Remove right
64         while(cur_r > qr) remove(a[cur_r--]);
65         // Remove left
66         while(cur_l < ql) remove(a[cur_l++]);
67
68         res[queries[i].id] = get_res();
69     }
70
71     for(int i = 1; i <= q; ++i)
72         cout << res[i] << "\n";
73 }

```

## 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  {
10     int cur_l = l / BLK;
11     int cur_r = r / BLK;
12     int ans = 0;
13
14     if(cur_l == cur_r) {
15         for (int i = l; i <= r; ++i)
16             ans += (a[i] >= val);
17     } else {
18         for(int i = l, _end = (cur_l + 1) * BLK; i < _end; ++i)
19             ans += (a[i] >= val);
20         for(int i = cur_l * BLK + 1; i <= cur_r * BLK; ++i)
21             ans += bs[i].end() - lower_bound(bs[i].begin(), bs[i].end(), val);
22         for(int i = cur_r * BLK + 1; i <= r; ++i)
23             ans += (a[i] >= val);
24     }
25     return ans;
26 }
27
28 void build()
29 {
30     for(int i = 0; i < n; ++i)
31         bs[i / BLK].emplace_back(a[i]);
32
33     for(int i = 0; i < M; ++i)
34         sort(bs[i].begin(), bs[i].end());
35 }
36
37 void update(int id, int delta)

```

```

38 {
39     int pos = lower_bound(bs[id / BLK].begin(), bs[id / BLK].end(), a[id]) - bs[id / BLK].begin();
40     bs[id / BLK][pos] = delta;
41     sort(bs[id / BLK].begin(), bs[id / BLK].end());
42     a[id] = delta;
43 }
44
45 void Solve()
46 {
47     cin >> n;
48     for(int i = 1; i <= n; ++i) cin >> a[i];
49
50     build();
51
52     cin >> q;
53     while(q--)
54     {
55         cin >> type >> x >> y;
56         if(type == 0)
57         {
58             cin >> z;
59             cout << query(x, y, z) << endl;
60         }
61         else
62             update(x, y);
63     }
64 }

```

## 8 Miscellaneous

### 8.1 Double comparison

```

1  bool approximatelyEqual(double a, double b, double epsilon)
2  {
3      return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
4  }
5
6  bool essentiallyEqual(double a, double b, double epsilon)
7  {
8      return fabs(a - b) <= ((fabs(a) > fabs(b) ? fabs(b) : fabs(a)) * epsilon);
9  }
10
11 bool definitelyGreaterThan(double a, double b, double epsilon)
12 {
13     return (a - b) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
14 }
15
16 bool definitelyLessThan(double a, double b, double epsilon)
17 {
18     return (b - a) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
19 }

```

### 8.2 Fast input output

```

1  /**
2   * Fast Input/Output method for C++:
3   * 1. cin(with sync_with_stdio(false) & cin.tie(nullptr)):
4   *    - /n = 5e6/ => 420ms
5   *    - /n = 1e7/ => 742ms
6   *    - ll:
7   *      - /n = 5e6/ => 895ms
8   *
9   * 2. read (using getchar()):
10  *    - int:
11  *      - /n = 5e6/ => 173ms
12  *      - /n = 1e7/ => 172ms
13  *    - ll:
14  *      - /n = 5e6/ => 340ms
15  */
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 }

```

## 8.3 Gcd & Lcm

```

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

```

## 8.4 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 ll binExp(ll a, ll b, ll p) {
8     ll res = 1;
9     while (b) {
10         if (b & 1ll)
11             res = res * a % p;
12         a = a * a % p;
13         b >>= 1;
14     }
15     return res;

```

```

16 }
17
18 // 2. Modular Multiplication
19
20 ll binMul(ll a, ll b, ll p) {
21     ll res = 0;
22     a %= p;
23     while (b) {
24         if (b & 1ll)
25             res = (res + a) % p;
26         a = (a + a) % p;
27         b >>= 1;
28     }
29     return res;
30 }
31
32 // 3. Modular Multiplicative Inverse
33
34 ll modInv(ll b, ll p) {
35     return binExp(b, p - 2, p); // Guaranteed that p is a Prime Number
36 }

```

## 8.5 Overloaded operators to accept 128 Bit integer

```

1 typedef __uint128_t    uil128;
2 typedef __int128       i128;
3
4 template <class T> string to_string(T x)
5 {
6     int sn = 1; if(x < 0) sn = -1, x *= sn; string s = "";
7     do { s = "0123456789"[x % 10] + s, x /= 10; } while(x);
8     return (sn == -1 ? "-" : "") + s;
9 }
10
11 auto str_to_int(string x)
12 {
13     uil128 ret = (x[0] == '-' ? 0 : x[0] - '0');
14     for(int i = 1; i < x.size(); ++i) ret = ret * 10 + (x[i] - '0');
15     return (x[0] == '-' ? -1 * (i128)ret : ret);
16 }
17
18 istream & operator >> (istream & in, i128 & i) noexcept { string s; in >> s; i = str_to_int(s); return in; }
19 ostream & operator << (ostream & os, const i128 i) noexcept { os << to_string(i); return os; }
20 istream & operator >> (istream & in, uil128 & i) noexcept { string s; in >> s; i = str_to_int(s); return in; }
21 ostream & operator << (ostream & os, const uil128 i) noexcept { os << to_string(i); return os; }

```

## 8.6 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>;

```

## 8.7 stress test

```

1 mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
2
3 /** 64-bit signed int Generator
4 **/
5 i64 int64(i64 a, i64 b) {
6     return uniform_int_distribution <i64> (a, b)(rng);
7 }
8
9 /** Customize your Generator depending on the input
10 **/
11 void gen () {

```

```

12  ofstream cout("input.in");
13  i32 t = 2;
14  cout << t << endl;
15
16  while (t--> 0) {
17      i32 n = int64(1, 100), m = int64(1, 100);
18      cout << n << " " << m << endl;
19
20      while (m--> 0) {
21          i32 u = int64(1, n), v = int64(1, n), c = int64(1, 4);
22          cout << u << " " << v << " " << c << endl;
23      }
24  }
25 }
26
27 i32 main (i32 arg, char* args[]) {
28     fast();
29
30     i32 tc = 0;
31     i32 limit = 100;
32     if(arg != 3) return 0;
33
34     string flags = "g++ -Wall -Wextra -Wshadow -Og -g -Ofast -std=c++17 -D_GLIBCXX_ASSERTIONS -DDEBUG -
        ggdb3 -fsanitize=address,undefined -fmax-errors=2 -o ";
35     string ex = ".cpp", bf, oz, pr;
36
37     bf = flags + args[1] + " " + args[1] + ex;
38     oz = flags + args[2] + " " + args[2] + ex;
39     char bff(bf.size() + 1);
40     char ozz(oz.size() + 1);
41     strcpy(bff, bf.c_str());
42     strcpy(ozz, oz.c_str());
43
44     // compile command
45     system(bff);

```

```

46     system(ozz);
47
48     ex = ".out";
49     pr = "./";
50     bf = pr + args[1] + " < input.in > " + args[1] + ex;
51     oz = pr + args[2] + " < input.in > " + args[2] + ex;
52     strcpy(bff, bf.c_str());
53     strcpy(ozz, oz.c_str());
54
55     while (++tc <= limit) {
56         gen();
57         cerr << tc << endl;
58         // run command
59         system(bff);
60         system(ozz);
61
62         ifstream brute_forces("brute_force.out");
63         ifstream optimizes("optimized.out");
64
65         string brute_force, optimized;
66         getline(brute_forces, brute_force, (char)EOF);
67         getline(optimizes, optimized, (char)EOF);
68
69         if(brute_force != optimized) {
70             cerr << "Wrong Answer" << endl;
71             break;
72         } else if (tc == limit) {
73             cout << "Accepted" << endl;
74         }
75     }
76 }

```