Elite Squad Team Reference Material

Contents

1 Setup

1.1 Vimrc

```
1  let mapleader = "\"
2
3  syntax on
4  filetype plugin on
5
6  set nocompatible
7  set autoread
8  set foldmethod=marker
9  set autoindent
8  set clipboard+=unnamedplus
11 set encoding=utf-8
12 set number relativenumber
13 set shiftwidth=2 softtabstop=2 expandtab
14
15 map <leader>c :w! && !compile %<CR>
16 vmap < <gv
17 vmap > >gv
18 nmap Y y$
```

1.2 Replace Capslock with 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
3  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}
```

2 Graph algorithms

2.1 Adjacency list

```
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

```
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

```
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 All paths 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 }
```

2.5 Articulation points and bridges

```
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 }
```

2.6 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      }
46      else if(To[i] != Par[node])
47        dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
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"[j == BiCCs[i].size() - 1];
65    }
66  }
```

2.7 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  }
```

2.8 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  }
```

2.9 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  }
```

2.10 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  }
```

2.11 DCG into DAG

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

```
12      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  }
```

2.12   Dijkstra [DG]

```
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  }
```

2.13   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      {
24          tie(cost, r, c) = Q.top(); Q.pop();
25          if((-cost) > dis[r][c]) continue; // lazy deletion
26
27          for(int i = 0; i < 4; ++i)
28          {
29          nr = r + dr[i];
30          nc = c + dc[i];
31
32          if(!valid(nr, nc)) continue;
33
34          if(dis[r][c] + grid[nr][nc] < dis[nr][nc])
35              {
36              dis[nr][nc] = dis[r][c] + grid[nr][nc];
37              Q.push({-dis[nr][nc], nr, nc});
38              }
39          }
40      }
41  }
```

2.14   Dijkstra [NWE]

```
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  }
```

2.15   Dijkstra [SG]

```
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  }
```

2.16   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
35    dfs_num[node] = VISITED;
36  }
37
38  int main() {
39    for(int i = 1; i <= n; ++i) if(!dfs_num[i])
```

```
40        edgeClassification(i);
41  }
```

2.17   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  }
```

2.18   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  }
```

2.19   Floyd warshall

```
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  }
```

2.20   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 minWieght = 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                      minWieght += cost;
19                      mstEdges.push_back(make_pair(from, to));
20                  }
21          }
22
23      if(mstEdges.size() == n - 1)
24          return make_pair(minWieght, mstEdges);
25
26      return make_pair(-1, vector < pair <int, int> > ());
27  }
```

2.21   Kth ancestor and LCA

```
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          {
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  }
```

2.22   LCA [Eulerian tour and RMQ]

```
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  }
```

2.23 Minimum vertex cover [Tree]

```
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  }
```

2.24 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 cycle

```
1   /** for each node run BFS and minmize 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  }
```

2.26 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
                queue).
5        - SPFA can deal with negative weight edge. If the graph has no negative cycle, SPFA runs well on
                it.
6        - If the graph has negative cycle(s), SPFA can also detect it as there must be some vertex (those
                on the negative cycle)
7        that enters the queue for over V dash 1 times.
8  **/
9
10 int Head[N], Par[N], Next[M], To[M], Cost[M], Cnt[N], ne, n, m, u, v, st, tax;
11 ll dis[N];
12 bool Inq[N];
13
14 void _set() {
15   memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
16   memset(Par,   -1, sizeof(Par[0]) * (n + 2));
17   memset(Cnt,    0, sizeof(Cnt[0]) * (n + 2));
18   memset(Inq,    0, sizeof(Inq[0]) * (n + 2));
19 }
20
21 bool SPFA(int src)
22 {
23   _set();
24
25   deque <int> Q;
26   Q.push_front(src);
27
28   dis[src] = 0;
29   Cnt[src] = 1;
30   Inq[src] = 1;
31
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 }
```

2.27 SPSP

```
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 BFS(int src, int trg)
5  {
6    memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
7    memset(Par,   -1, sizeof(Par[0]) * (n + 2));
8
9    deque <int> Q;
10   Q.push_front(src);
11   dis[src] = 0;
12
13   int node;
14   while(Q.size()) {
15     node = Q.front(); Q.pop_front();
16     if(node == trg) return;
17
18     for(int i = Head[node]; i; i = Next[i])
19       if(dis[node] + Cost[i] < dis[To[i]]) {
20 dis[To[i]] = dis[node] + Cost[i];
21 if(Cost[i])
22   Q.push_back(To[i]);
23 else
24   Q.push_front(To[i]);
25       }
26   }
27 }
```

2.28 SPSP [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 /**
15    7 0 1
16    \|/
17   6-*-2
18    /|\
19    5 4 3
20 **/
21
22 int ZBFS(int sr, int sc, int tr, int tc)
23 {
24   memset(dis, 0x3f, sizeof (dis)); // memset(dis, 0x3f, n * m) we don't do that here
25
26   deque <pair <int, int> > Q;
27
28   dis[sr][sc] = 0;
29   Q.push_front({sr, sc});
30
31   int r, c, nr, nc, ncost;
32   while(Q.size()) {
33     tie(r, c) = Q.front(); Q.pop_front();
34     if(r == tr && c == tc) return dis[r][c];
35
36     for(int i = 0; i < 8; ++i) {
37       nr = r + dr[i];
38       nc = c + dc[i];
39
40       if(!valid(nr, nc)) continue;
41       ncost = (i != grid[r][c]);
42
43       if(dis[r][c] + ncost < dis[nr][nc]) {
44 dis[nr][nc] = dis[r][c] + ncost;
45
46 if(ncost)
47   Q.push_back({nr, nc});
48 else
49   Q.push_front({nr, nc});
50       }
51     }
52   }
53   return oo;
54 }
```

2.29 SSSP

```
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 }
```

2.30   SSSP [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  }
```

2.31   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   Tarjan

```
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  }
```

2.33   Tree diameter

```
1   int Head[N], Next[M], To[M], Par[N], toLeaf[N], maxLength[N], ne, n, m, u, v, w;
2
3   void _clear() {
4     memset(Head, 0, sizeof(Head[0]) * (n + 2));
5     memset(Par, -1, sizeof(Par[0])  * (n + 2));
6     ne = 0;
7   }
8
9   void dfs_toLeaf(int node, int par = -1)
10  {
11    toLeaf[node] = 0;
12    for(int i = Head[node]; i; i = Next[i]) if(To[i] != par) {
13      dfs_toLeaf(To[i], node);
14      if(toLeaf[To[i]] + 1 > toLeaf[node])
15        toLeaf[node] = toLeaf[To[i]] + 1;
16    }
17  }
18
19  void dfs_maxLength(int node, int par = -1)
20  {
21    int firstMax = -1;
22    int secondMax = -1;
23    for(int i = Head[node]; i; i = Next[i]) if(To[i] != par) {
24      dfs_maxLength(To[i], node);
25
26      if(toLeaf[To[i]] > firstMax) {
27        if(firstMax > secondMax)
28          secondMax = firstMax;
29        firstMax = toLeaf[To[i]];
30      } else if(toLeaf[To[i]] > secondMax)
31        secondMax = toLeaf[To[i]];
32    }
33    maxLength[node] = firstMax + secondMax + 2;
34  }
35
36  void main()
37  {
38    dfs_toLeaf(1);
39    dfs_maxLength(1);
40
41    int diameter = 0;
42    for(int i = 1; i <= n; ++i)
43      if(maxLength[i] > diameter)
44        diameter = maxLength[i];
45
46    cout << diameter << endl;
47  }
```

2.34   Tree diameter [Weighted DFS]

```
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, 0ll);
10   from = at, diameter = 0;
11   DFS(from, 0ll);
12   to = at;
13   cout << diameter << endl;
14 }
```

2.35   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
55   SP[i][j] = H[x] <= H[y] ? x : y;
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 }
```

2.36   TS [Kahns 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 }
```

3   Data structures

3.1   Merge sort tree

```
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 };
```

3.2 Sparse table RMQ

```
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
                                                 = O(n log n)
41        {
42            k = (1 << j);
43            d = (k >> 1);
44
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
55    T query(int l, int r) // this query is O(1)
56    {
57        int d = r - l + 1;
58        T const & x = ST[l][Log[d]];
59        T const & y = ST[l + d - (1 << Log[d])][Log[d]];
60
61        return func(_A[x], _A[y]) ? x : y;
62    }
63 };
```

3.3 Sparse table RSQ

```
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, F _func = [](T a, T b) { return a + b; }) : 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] = _A[i];
39
40      for(int j = 1, k, d; j <= _LOG; ++j)
41          {
42  k = (1 << j);
43  d = (k >> 1);
44
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(x, y);
51      }
52      }
53  }
54
55  T query(int l, int r)
56  {
57      int d = r - l + 1;
58      T ret = 0;
59
60      for(int i = l; d; i += lastBit(d), d -= lastBit(d))
61          ret = func(ret, ST[i][Log[lastBit(d)]]);
62
63      return ret;
64  }
65
66  int lastBit(int a) {
67      return (a & -a);
68  }
69  };
```

3.4   Segment tree RMQ

```
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(
            _func)
13      {
14          _N = distance(_begin, _end);
15          _N = (1 << (int)ceil(log2(_N)));
16
17          _A.assign(_N + 1,  0);
18          ST.assign(_N << 1, 0);
19          LT.assign(_N << 1, 0);
20
21          __typeof(_begin) i = _begin;
22          for(int j = 1; i != _end; ++i, ++j)
23              _A[j] = *i;
24
25          build(1, 1, _N);
26      }
27
28      void build(int p, int l, int r)
29      {
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 };
```

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       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  {
24    vector <int> par;
25    vector <int> siz;
26    int num_sets;
27    size_t sz;
28
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 Segment tree RSQ

```
 1  class SegmentTree
 2  {
 3    vector <ll> sTree;
 4    vector <ll> lazyTree;
 5    vector <int> localArr;
 6    int NP2, oo = 0x3f3f3f3f;
 7    ll INF = 0x3f3f3f3f3f3f3f3f;
 8
 9  public :
10    template <class T>
11    SegmentTree(T _begin, T _end)
12    {
13      NP2 = 1;
14      int n = _end - _begin;
15      while(NP2 < n) NP2 <<= 1;
16
17      sTree.assign(NP2 << 1, 0);
18      lazyTree.assign(NP2 << 1, 0);
19      localArr.assign(NP2 + 1, 0);
20
21      __typeof(_begin) i = _begin;
22      for(int j = 1; i != _end; i++, ++j)
23        localArr[j] = *i;
24
25      build(1, 1, NP2);
26    }
27
28    void build(int p, int l, int r)
29    {
30      if(l == r) {
31        sTree[p] = localArr[l];
32        return;
33      }
34
35      build(left(p),  l,      mid(l, r));
36      build(right(p), mid(l, r) + 1, r);
37
38      sTree[p] = sTree[left(p)] + sTree[right(p)];
39    }
40
41    void update_point(int inx, int delta)
42    {
43      inx += NP2 - 1;
44      sTree[inx] += delta;
45
46      while(inx > 1) {
47        inx >>= 1;
48        sTree[inx] = sTree[left(inx)] + sTree[right(inx)];
49      }
50    }
51
52    void update_range(int ul, int ur, int delta) {
53      update_range(ul, ur, delta, 1, 1, NP2);
54    }
55
56    ll query(int ql, int qr) {
57      return query(ql, qr, 1, 1, NP2);
58    }
59
60  private :
61    void update_range(int ul, int ur, int delta, int p, int l, int r)
62    {
63      if(isOutside(ul, ur, l, r))
64        return;
65
66      if(isInside(ul, ur, l, r)) {
67        sTree[p] += (r - l + 1) * 1ll * delta;
68        lazyTree[p] += delta;
69        return;
70      }
71
72      propagate(p, l, r);
73
74      update_range(ul, ur, delta, left(p),  l,      mid(l, r));
75      update_range(ul, ur, delta, right(p), mid(l, r) + 1, r);
76
77      sTree[p] = sTree[left(p)] + sTree[right(p)];
78    }
79
80    ll query(int ql, int qr, int p, int l, int r)
81    {
82      if(isOutside(ql, qr, l, r))
83        return 0;
84
85      if(isInside(ql, qr, l, r))
86        return sTree[p];
```

```
87        }
88
89      propagate(p, l, r);
90
91      return query(ql, qr, left(p),   l,     mid(l, r)) +
92             query(ql, qr, right(p), mid(l, r) + 1, r);
93    }
94
95    void propagate(int p, int l, int r)
96    {
97      if(lazyTree[p]) {
98        sTree[left(p)]  += (mid(l, r) - l + 1) * 1ll * lazyTree[p];
99        sTree[right(p)] += (r - mid(l, r)) * 1ll * lazyTree[p];
100
101       if(l != r) {
102   lazyTree[left(p)]  += lazyTree[p];
103   lazyTree[right(p)] += lazyTree[p];
104       }
105       lazyTree[p] = 0;
106     }
107   }
108
109   inline bool isInside(int ql, int qr, int sl, int sr) {
110     return (ql <= sl && sr <= qr);
111   }
112
113   inline bool isOutside(int ql, int qr, int sl, int sr) {
114     return (sr < ql || qr < sl);
115   }
116
117   inline int mid (int l, int r) {
118     return ((l + r) >> 1);
119   }
120
121   inline int left(int p) {
122     return (p << 1);
123   }
124
125   inline int right(int p) {
126     return ((p << 1) | 1);
127   }
128 };
```

3.7   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 = l, 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 }
```

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   ui128 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; 1ll * 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 <i128> C(ra, vector <i128> (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] + (i128)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 i128 ModExp(i128 a, ll p) {
105   i128 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   cin >> n;
178   vector < array <ll, 2> > factors = factorize(n);
179
180   i128 ans = 1;
181   for (int i = 0; i < (int)factors.size(); ++i) {
182     ans = lcm(ans, (i128)pisano_prime(factors[i][0]) * ModExp(factors[i][0], factors[i][1] - 1));
183   }
184   cout << ans << endl;
185 }
186
187 void MultiTest(bool Tests)
188 {
189   int tc = 1; (Tests) && (cin >> tc);
190   for(int i = 1; i <= tc; ++i)
191     Solve();
192 }
193
194 int main()
195 {
196   Fast(); initialize(); MultiTest(1);
197 }
```

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 <= 1e{14}
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 <= 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
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(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
28  ll phi_factorial(int n) {
29    ll ret = 1;
30    for (int i = 2; i <= n; ++i) {
31      ret = ret * (lp[i] == i ? 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   #pragma  GCC optimize ("Ofast")
2
3   #include <bits/stdc++.h>
4
5   #define endl    '\n'
6
7   using namespace std;
8
9   typedef long long  ll;
10  typedef __int128 i128;
11
12  const int N = 1e6;
13
14  void Fast() {
15    cin.sync_with_stdio(0);
16    cin.tie(0);cout.tie(0);
17  }
18
19  ll ModExp(ll base, ll e, ll mod)
20  {
21    ll result;
22    base %= mod;
23
24    for(result = 1; e; e >>= 1ll)
25      {
26        if(e & 1ll)
27  result = ((i128)result * base) % mod;
28        base = ((i128)base * base) % mod;
29      }
30    return result;
31  }
32
33  bool CheckComposite(ll n, ll p, ll d, int r)
34  {
35    ll a = ModExp(p, d, n);
36    if(a == 1 || a == n - 1)
37      return false;
```

```
38
39    for(int i = 1; i < r; ++i)
40      {
41        a = ((i128)a * a) % n;
42        if(a == n - 1)
43    return false;
44      }
45    return true;
46  }
47
48  bool Miller(ll n)
49  {
50    if(n < 2) return false;
51
52    int r; ll d;
53    for(r = 0, d = n - 1; (d & 1ll) == 0; d >>= 1ll, ++r);
54
55    for(int p : {2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 37})
56      {
57        if(n == p)
58    return true;
59        if(CheckComposite(n, p, d, r))
60    return false;
61      }
62    return true;
63  }
64
65  int main()
66  {
67    Fast();
68
69    ll n;
70    cin >> n;
71    cout << (Miller(n) ? "Yes, it is Prime" : "No, it is not a prime") << endl;
72  }
```

4.10   Stable marriage problem

```
1   #include <bits/stdc++.h>
2
3   #define endl    '\n'
4
5   using namespace std;
6
7   const int N = 40;
8   int n;
9   struct woman {
10    int husband, pref_list[N];
11    char name;
12    woman () {
13      memset(pref_list, 0x00, sizeof pref_list);
14      husband = 0;
15      name = '\0';
16    }
17  };
18
19  struct man {
20    int next_proposal, pref_list[N];
21    char name;
22    man () {
23      memset(pref_list, 0x00, sizeof pref_list);
24      next_proposal = 1;
25      name = '\0';
26    }
27  };
28
29  char u, v, why;
30  map <char, int> mp;
31  queue <int> single;
32  vector < array <char, 2> > matching_list;
33  man men[N];
34  woman women[N];
35
36  void _clear() {
37    mp.clear();
38    single = queue <int> ();
39    matching_list = vector < array <char, 2> > ();
40  }
41
42  void Solve()
43  {
44    cin >> n;
45    _clear();
46
47    for(int i = 1; i <= n; ++i) {
48      cin >> u, mp[u] = i;
49      men[i] = man();
```

```
50      men[i].name = u;
51      single.push(i);
52    }
53    for(int i = 1; i <= n; ++i) {
54      cin >> v, mp[v] = i;
55      women[i] = woman();
56      women[i].name = v;
57    }
58    for(int i = 1; i <= n; ++i) {
59      cin >> u >> why;
60      for(int j = 1; j <= n; ++j) {
61        cin >> v;
62        men[mp[u]].pref_list[j] = mp[v];
63      }
64    }
65    for(int i = 1; i <= n; ++i) {
66      cin >> v >> why;
67      for(int j = 1; j <= n; ++j) {
68        cin >> u;
69        women[mp[v]].pref_list[mp[u]] = j;
70      }
71    }
72
73    int cur_man, cur_woman, ex_man;
74    while(!single.empty()) {
75      cur_man = single.front();
76      cur_woman = men[cur_man].pref_list[men[cur_man].next_proposal];
77
78      if(women[cur_woman].husband == 0) {
79        women[cur_woman].husband = cur_man;
80        single.pop();
81      } else if(women[cur_woman].pref_list[cur_man] < women[cur_woman].pref_list[women[cur_woman].
              husband]) {
82        ex_man = women[cur_woman].husband;
83        women[cur_woman].husband = cur_man;
84        single.pop();
85        single.push(ex_man);;
86      }
87      ++men[cur_man].next_proposal;
88    }
89
90    for(int i = 1; i <= n; ++i)
91      matching_list.push_back({men[women[i].husband].name, women[i].name});
92
93    sort(matching_list.begin(), matching_list.end());
94
95    for(array <char, 2> p : matching_list)
96      cout << p[0] << " " << p[1] << endl;
97  }
98
99  void MultiTest(bool Tests = 0)
100 {
101   int tc = 1; (Tests) && (cin >> tc);
102   for(int i = 1; i <= tc; ++i) {
103     if(i > 1) cout << endl;
104     Solve();
105   }
106 }
107
108 /**------------------------>>  Main  <<------------------------**/
109
110 int main()
111 {
112   MultiTest(1);
113 }
```

4.11   Euler phi

```
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

```
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 }
```

5   String Processing

5.1   Trie

```
1  class Trie {
2  private:
3    Trie* children[26]; // Pointer = 8 Byte; 8*26 = 208 Byte
4    int prefixs, 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>>>
          occurrence; // small at top
9
10 public:
11   Trie(char lett = '\0') {
12     memset(children, 0, sizeof(children));
13     prefixs = 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) {
```

（side）Al-Azhar University

```
 39        return 0;
 40      }
 41      cur = cur->children[inx];
 42    }
 43    return cur->words;
 44  }
 45
 46  int search_prefix(string &str) { // O(l)
 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
                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   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]; // faill 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   Misc Topics

7.1   A*-Algorithm

```
1   #pragma GCC optimize("Ofast")
2
3   #include <bits/stdc++.h>
4
5   #define endl    '\n'
6
7   using namespace std;
8
9   typedef int64_t  ll;
10
11  void Fast() {
12      cin.sync_with_stdio(0);
13      cin.tie(0);cout.tie(0);
14  }
15
16  const int dr []     = {-1, 0, 1, 0};
17  const int dc []     = {0, 1, 0, -1};
18  const char dir []   = {'U', 'R', 'D', 'L'};
19  map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};
20
21  const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
22  const ll INF = 0x3f3f3f3f3f3f3f3f;
23
24  char grid[N][N];
25  int dis[N][N], n, m, si, sj, ti, tj;
26  char Par[N][N];
27
28  vector < pair <int, int> > restorePath(int sr, int sc, int tr, int tc)
29  {
30      vector < pair <int, int> > ret;
31      if(dis[tr][tc] == oo) return ret;
32
33      for(char i = Par[tr][tc]; (sr ^ tr) || (sc ^ tc); i = Par[tr][tc])
34      {
35          ret.push_back({tr, tc});
36          tr += dr[inv[i]];
37          tc += dc[inv[i]];
38      }
39
40      ret.push_back({sr, sc});
41      reverse(ret.begin(), ret.end());
42      return ret;
43  }
44
45  bool valid(int r, int c) {
46      return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] != '%';
47  }
```

```
48
49  /** admissible heuristic **/
50  int manhattanDistance(int x1, int y1, int x2, int y2) {
51      return (abs(x1 - x2) + abs(y1 - y2));
52  }
53
54  int Astar(int sr, int sc, int tr, int tc)
55  {
56      memset(dis, 0x3f, sizeof (dis));
57      memset(Par,   -1, sizeof (Par));
58
59      priority_queue <tuple <int, int, int> > Q;
60
61      dis[sr][sc] = 0;
62      Q.push({-manhattanDistance(sr, sc, tr, tc), sr, sc});
63
64      int hcost, r, c, nr, nc;
65      while(Q.size())
66      {
67          tie(hcost, r, c) = Q.top(); Q.pop();
68          if(r == tr && c == tc) return dis[r][c];
69
70          for(int i = 0; i < 4; ++i)
71          {
72              nr = r + dr[i];
73              nc = c + dc[i];
74
75              if(!valid(nr, nc)) continue;
76
77              if(dis[r][c] + 1 < dis[nr][nc])
78              {
79                  dis[nr][nc] = dis[r][c] + 1;
80                  Par[nr][nc] = dir[i ^ 2];
81                  Q.push({-dis[nr][nc] -manhattanDistance(nr, nc, tr, tc), nr, nc});
82              }
83          }
84      }
85      return -1;
86  }
87
88  void Solve()
89  {
90      Fast();
91
92      cin >> si >> sj >> ti >> tj >> n >> m;
93
94      for(int i = 0; i < n; ++i)
95          for(int j = 0; j < m; ++j)
96              cin >> grid[i][j];
97
98      cout << Astar(si, sj, ti, tj) << endl;
99      vector < pair <int, int> > path = restorePath(si, sj, ti, tj);
100
101     for(auto point : path)
102         cout << point.first << " " << point.second << endl;
103 }
104
105 int main()
106 {
107     int t = 1;
108     while(t--) Solve();
109 }
110
111 /**
112     P -> strat
113     . -> target
114
115     input:
116     0 2 2 3 5 5
117     %%P%-
118     -%---
119     %--.-
120     %%%%%
121     -----
122
123     output:
124     3
125     0 2
126     1 2
127     2 2
128     2 3
129
130 **/
```

7.2   Mo's algorithm

```
1   #pragma GCC optimize ("Ofast")
```

```
2
3    #include <bits/stdc++.h>
4
5    #define endl        '\n'
6
7    using namespace std;
8
9    typedef int64_t    ll;
10   typedef __int128 i128;
11
12   void Fast() {
13     cin.sync_with_stdio(0);
14     cin.tie(0);cout.tie(0);
15   }
16
17   void File() {
18     freopen("input.in",  "r", stdin);
19     freopen("output.out", "w", stdout);
20   }
21
22   const int N = 3e4 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
23   const ll INF = 0x3f3f3f3f3f3f3f3f;
24   const int BLK = 256;
25
26   struct query
27   {
28     int l, r, id, blk;
29
30     query() = default;
31     query(int _l, int _r, int _id) {
32       l = _l;
33       r = _r;
34       id = _id;
35       blk = l / BLK;
36     }
37
38     bool operator < (const query other) const {
39       if(blk ^ other.blk)
40         return blk < other.blk;
41       return (blk & 1) ? r < other.r : r > other.r;
42     }
43   } queries[M];
44
45   int res[M], freq[M << 3], cur;
46
47   void add(int id) {
48     cur += (++freq[id] == 1);
49   }
50
51   void remove(int id) {
52     cur -= (--freq[id] == 0);
53   }
54
55   int get_res() {
56     return cur;
57   }
58
59   int cur_l, cur_r, l, r, n, q, a[N];
60
61   void Solve()
62   {
63     cin >> n;
64     for(int i = 1; i <= n; ++i) cin >> a[i];
65
66     cin >> q;
67     for(int i = 1; i <= q; ++i) {
68       cin >> l >> r;
69       queries[i] = query(l, r, i);
70     }
71
72     sort(queries + 1, queries + 1 + q);
73
74     cur_l = 1, cur_r = 0; // assign to right invalid index
75     for(int i = 1; i <= q; ++i)
76       {
77         int ql = queries[i].l;
78         int qr = queries[i].r;
79
80         // Add right
81         while(cur_r < qr) add(a[++cur_r]);
82         // Add left
83         while(cur_l > ql) add(a[--cur_l]);
84         // Remove right
85         while(cur_r > qr) remove(a[cur_r--]);
86         // Remove left
87         while(cur_l < ql) remove(a[cur_l++]);
88
89         res[queries[i].id] = get_res();
90       }
91
92     for(int i = 1; i <= q; ++i)
93       cout << res[i] << "\n";
```

```
94   }
95
96   int main()
97   {
98     Fast();
99
100    int tc = 1;
101    for(int i = 1; i <= tc; ++i)
102      Solve();
103  }
```

7.3   SQRT decomposition

```
1    #pragma GCC optimize ("Ofast")
2
3    #include <bits/stdc++.h>
4
5    #define endl        '\n'
6
7    using namespace std;
8
9    typedef int64_t    ll;
10   typedef __int128 i128;
11
12   void Fast() {
13     cin.sync_with_stdio(0);
14     cin.tie(0);cout.tie(0);
15   }
16
17   const int N = 5e5 + 9, M = 1e3 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
18   const ll INF = 0x3f3f3f3f3f3f3f3f;
19   const int BLK = 256;
20
21   int n, q, a[N], type, x, y, z;
22   vector <int> bs[M];
23
24   int query(int l, int r, int val)
25   {
26     int cur_l = l / BLK;
27     int cur_r = r / BLK;
28     int ans = 0;
29
30     if(cur_l == cur_r) {
31       for (int i = l; i <= r; ++i)
32         ans += (a[i] >= val);
33     } else {
34       for(int i = l, _end = (cur_l + 1) * BLK; i < _end; ++i)
35         ans += (a[i] >= val);
36       for(int i = cur_l + 1; i <= cur_r - 1; ++i)
37         ans += bs[i].end() - lower_bound(bs[i].begin(), bs[i].end(), val);
38       for(int i = cur_r * BLK; i <= r; ++i)
39         ans += (a[i] >= val);
40     }
41     return ans;
42   }
43
44   void build()
45   {
46     for(int i = 0; i < n; ++i)
47       bs[i / BLK].emplace_back(a[i]);
48
49     for(int i = 0; i < M; ++i)
50       sort(bs[i].begin(), bs[i].end());
51   }
52
53   void update(int id, int delta)
54   {
55     int pos = lower_bound(bs[id / BLK].begin(), bs[id / BLK].end(), a[id]) - bs[id / BLK].begin();
56     bs[id / BLK][pos] = delta;
57     sort(bs[id / BLK].begin(), bs[id / BLK].end());
58     a[id] = delta;
59   }
60
61   void Solve()
62   {
63     cin >> n;
64     for(int i = 1; i <= n; ++i) cin >> a[i];
65
66     build();
67
68     cin >> q;
69     while(q--)
70       {
71         cin >> type >> x >> y;
72         if(type == 0)
73           {
74             cin >> z;
```

```
75        cout << query(x, y, z) << endl;
76            }
77          else
78      update(x, y);
79      }
80  }
81
82  int main()
83  {
84      Fast();
85
86      int tc = 1;
87      for(int i = 1; i <= tc; ++i)
88          Solve();
89  }
```

8   Misc

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 IO

```
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      if (minus) return -(ll)result;
39      return result;
40  }
41
42  int readi () {
43      bool minus = false;
44      unsigned int result = 0;
45      char ch;
46      ch = getchar();
47
48      while (true) {
49          if (ch == '-') break;
50          if (ch >= '0' && ch <= '9') break;
51          ch = getchar();
52      }
53
54      if (ch == '-') minus = true;
55      else result = ch - '0';
56
57      while (true) {
58          ch = getchar();
59          if (ch < '0' || ch > '9') break;
60          result = result * 10 + (ch - '0');
61      }
62
63      if (minus) return -(int)result;
64      return result;
65  }
```

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
            with any operations that have the property of associativity:
3   */
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        ui128;
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    ui128 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, ui128 & i) noexcept { string s; in >> s; i = str_to_int(s);
        return in; }
21 ostream & operator << (ostream & os, const ui128 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  #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 -
        ggdb3 -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[bf.size() + 1];
63    char ozz[oz.size() + 1];
64    strcpy(bff, bf.c_str());
65    strcpy(ozz, 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" << endl;
97       }
98    }
99 }
```