# ICPC Team Reference Material

# Contents


**1 Setup** 1
- 1.1 Vimrc 1
- 1.2 Capslock as Escape 1
- 1.3 Compilation 1

**2 Graph algorithms** 1
- 2.1 Adjacency list representation 1
- 2.2 Depth first search (DFS) 1
- 2.3 Breadth first search (BFS) 1
- 2.4 0-1 BFS 1
- 2.5 0-1 BFS (grid) 2
- 2.6 Articulation points and bridges 2
- 2.7 Bi-connected components 2
- 2.8 Bipartite graph 3
- 2.9 Bellman ford 3
- 2.10 Connected components 3
- 2.11 Cycle detection 3
- 2.12 Directed cyclic graph into acyclic 3
- 2.13 Dijkstra (dense graph) 4
- 2.14 Dijkstra (grid) 4
- 2.15 Dijkstra (negative weighted graph) 4
- 2.16 Dijkstra (sparse graph) 4
- 2.17 Edge classification 5
- 2.18 Eulerian tour tree 5
- 2.19 Floodfill 5
- 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) 6
- 2.24 Minimum vertex cover 7
- 2.25 Shortest path faster algorithml (SPFA) 7
- 2.26 Single source shortest path 7
- 2.27 Single source shortest path (grid) 7
- 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) 8
- 2.32 Problems 8
  - 2.32.1 Restoring the path 8
  - 2.32.2 Tree distances 9
  - 2.32.3 Subtree sizes 9
  - 2.32.4 Shortest cycle 9
  - 2.32.5 Path sum for each node 10

**3 Data structures** 10
- 3.1 Segment tree 10
- 3.2 Merge sort tree 11
- 3.3 Merge sort 11
- 3.4 Sparse table 11
- 3.5 Union find disjoint sets 12
- 3.6 Bubble sort 12

**4 Mathematics** 13
- 4.1 Pisano periodic sequence 13
- 4.2 Euler totient function 14
- 4.3 Extended wheel factorization 14
- 4.4 Least prime factorization 14
- 4.5 Mobius function 14
- 4.6 Phi factorial 15
- 4.7 Linear sieve 15
- 4.8 Segmented sieve 15
- 4.9 Miller Rabin test 15
- 4.10 Stable marriage problem 16
- 4.11 Euler totient sieve 16
- 4.12 Mobius sieve 16
- 4.13 Wheel sieve 17

**5 String Processing** 17
- 5.1 Trie 17
- 5.2 Knuth Morris Pratt (KMP) 18

**6 Geometry** 18
- 6.1 Point 18

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

**8 Miscellaneous** 20
- 8.1 Double comparison 20
- 8.2 Fast input output 21
- 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 22
- 8.7 stress test 22


# 1 Setup

## 1.1 Vimrc

```
let mapleader = "\"
syntax on
filetype plugin on
set nocompatible
set autoread
set foldmethod=marker
set autoindent
set clipboard+=unnamedplus
set number relativenumber
set shiftwidth=2 softtabstop=2 expandtab
map <leader>c :w! && !compile %:p:r<CR>
vmap < <gv
vmap > >gv
```

## 1.2 Capslock as Escape

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

## 1.3 Compilation

```
#!/bin/bash
# put this file in .local/bin or add its dir to the PATH variable
compile() {
  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}
}
compile "$1"
```

# 2 Graph algorithms

## 2.1 Adjacency list representation

```
class Graph {
public:
  vector <int> _head, _next, _to, _cost;
  int edge_number;
  bool isDirected;

  Graph() = default;
  Graph(int V, int E, bool isDirec) {
    isDirected = isDirec;
    _head.assign(V + 9, 0);
    _next.assign(isDirected ? E + 9 : E * 2 + 9, 0);
    _to.assign(isDirected ? E + 9 : E * 2 + 9, 0);
    // _cost.assign(isDirected ? E + 9 : E * 2 + 9, 0);
    edge_number = 0;
  }

  void addEdge(int u, int v, int w = 0) {
    _next[++edge_number] = _head[u];
    _to[edge_number] = v;
    // _cost[edge_number] = w;
    _head[u] = edge_number;
  }

  void addBiEdge(int u, int v, int w = 0) {
    addEdge(u, v, w);
    addEdge(v, u, w);
  }

  void dfs(int node) {
    vis[node] = true;
    for(int i = _head[node]; i; i = _next[i]) if(!vis[_to[i]]) {
  dfs(_to[i]);
      }
  }
};
```

## 2.2 Depth first search (DFS)

```
void DFS(int node)
{
  vis[node] = true;
  for(int i = Head[node]; i; i = Next[i])
    if(!vis[To[i]])
      DFS(To[i]);
}
```

## 2.3 Breadth first search (BFS)

```
void BFS(int src)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  queue <int> q;
  q.push(src);
  dis[src] = 0;

  int u;
  while(q.size())
    {
      u = q.front(); q.pop();
      for(int i = Head[u]; i; i = Next[i]) if(dis[To[i]] == oo) {
    dis[To[i]] = dis[u] + 1;
    Par[To[i]] = u;
    q.push(To[i]);
        }
    }
}
```

## 2.4 0-1 BFS

```
#include <bits/stdc++.h>
using namespace std;
typedef int64_t  ll;
int main() {}

const int N = 1e5 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
ll INF = 0x3f3f3f3f3f3f3f3f;

int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void addEdge(int from, int to, int cost) {
    Next[++ne] = Head[from];
    Head[from] = ne;
    Cost[ne] = cost;
    To[ne] = to;
}

void _clear() {
    memset(Head, 0, sizeof(Head[0]) * (n + 2));
    ne = 0;
}

void BFS(int src, int trg)
{
    memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
    memset(Par,   -1, sizeof(Par[0]) * (n + 2));

    deque <int> Q;
    Q.push_front(src);
    dis[src] = 0;

    int node;
    while(Q.size())
    {
        node = Q.front(); Q.pop_front();
        if(node == trg) return;

        for(int i = Head[node]; i; i = Next[i])
            if(dis[node] + Cost[i] < dis[To[i]])
            {
                dis[To[i]] = dis[node] + Cost[i];
                if(Cost[i])
                    Q.push_back(To[i]);
                else
                    Q.push_front(To[i]);
            }
    }
}
```

## 2.5 0-1 BFS (grid)

```
#include <bits/stdc++.h>
using namespace std;
typedef int64_t  ll;
int main() {}

const int dr[]   = { -1, -1, 0, 1, 1,  1,  0, -1 };
const int dc[]   = {  0,  1, 1, 1, 0, -1, -1, -1 };
const char dir[] = {'D', 'U', 'R', 'L'};

const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;

int dis[N][N], n, m, si, sj, ti, tj;
char grid[N][N];

bool valid(int r, int c) {
    return r >= 1 && r <= n && c >= 1 && c <= m;
}

/**
7 0 1
 \|/
6-*-2
 /|\
5 4 3
**/

int ZBFS(int sr, int sc, int tr, int tc)
{
    memset(dis, 0x3f, sizeof (dis)); // memset(dis, 0x3f, n * m) we don't do that here

    deque <pair <int, int> > Q;

    dis[sr][sc] = 0;
    Q.push_front({sr, sc});

```

```
    int r, c, nr, nc, ncost;
    while(Q.size())
    {
        tie(r, c) = Q.front(); Q.pop_front();
        if(r == tr && c == tc) return dis[r][c];

        for(int i = 0; i < 8; ++i)
        {
            nr = r + dr[i];
            nc = c + dc[i];

            if(!valid(nr, nc)) continue;
            ncost = (i != grid[r][c]);

            if(dis[r][c] + ncost < dis[nr][nc])
            {
                dis[nr][nc] = dis[r][c] + ncost;

                if(ncost)
                    Q.push_back({nr, nc});
                else
                    Q.push_front({nr, nc});
            }
        }
    }
    return oo;
}
```

## 2.6 Articulation points and bridges

```
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;
bool Art[N];
vector < pair <int, int> > bridges(M);

void _clear() {
  memset(Head,    0, sizeof(Head[0])    * (n + 2));
  memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));
  memset(Par,    -1, sizeof(Par[0])     * (n + 2));
  memset(Art,     0, sizeof(Art[0])     * (n + 2));
  ne = dfs_timer = bridgeInx = 0;
}

void Tarjan(int node) {
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0) {
      if(node == root) ++rootChildren;
      Par[To[i]] = node;
      Tarjan(To[i]);
      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);

      if(dfs_low[To[i]] >= dfs_num[node])
  Art[node] = true;

      if(dfs_low[To[i]] > dfs_num[node])
  bridges[bridgeInx++] = make_pair(node, To[i]);
    }
    else if(To[i] != Par[node])
      dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
  }
}

int main() {
  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0) {
      root = i;
      rootChildren = 0;
      Tarjan(i);
      Art[root] = (rootChildren > 1);
    }

  cout << "Art Points :\n";
  for(int i = 1; i <= n; ++i)
    if(Art[i])
      cout << i << " ";

  cout << "\nBridges :\n";
  for(int i = 0; i < bridgeInx; ++i)
    cout << bridges[i].first << " - " << bridges[i].second << endl;
}
```

## 2.7 Bi-connected components

```
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;
bool Art[N];
vector < vector <int> > BiCCs(N), BiCCIDs(N);

void addEdge(int from, int to) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  To[ne] = to;
}

void _clear() {
  memset(Head,       0, sizeof(Head[0])       * (n + 2));
  memset(dfs_num,    0, sizeof(dfs_num[0])    * (n + 2));
  memset(Par,       -1, sizeof(Par[0])        * (n + 2));
  memset(Art,        0, sizeof(Art[0])        * (n + 2));
  ne = dfs_timer = top = ID = 0;
  BiCCs = BiCCIDs = vector < vector <int> > (N);
}

void Tarjan(int node)
{
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  Stack[top++] = node;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0) {
      if(node == root) ++rootChildren;
      Par[To[i]] = node;
      Tarjan(To[i]);

      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
      if(dfs_low[To[i]] >= dfs_num[node])
  {
    Art[node] = true;
    ++ID;
    for(int x = -1; x ^ To[i];)
      {
        x = Stack[--top];
        BiCCIDs[x].emplace_back(ID);
        BiCCs[ID].emplace_back(x);
      }
    BiCCIDs[node].emplace_back(ID);
    BiCCs[ID].emplace_back(node);
  }
    }
    else if(To[i] != Par[node])
      dfs_low[node] = Min(dfs_low[node], dfs_num[To[i]]);
  }
}

int main()
{
  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0) {
      root = i;
      rootChildren = 0;
      Tarjan(i);
      Art[root] = (rootChildren > 1);
    }

  for(int i = 1; i <= ID; ++i) {
    cout << "Component : " << i << " contains : ";
    for(int j = 0; j < (int)BiCCs[i].size(); ++j)
      cout << BiCCs[i][j] << " \n"[j == BiCCs[i].size() - 1];
  }
}
```

## 2.8 Bipartite graph

```
bool checkBiPartite(int node, int par = 0) {
  if(vis[node])
    return color[par] != color[node];

  color[node] = color[par] ^ 1;
  vis[node] = true;
  bool ok = true;
  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par)
      ok &= checkBiPartite(To[i], node);
  return ok;
}
```

```
int main() {
  bool isBiPartite = true;
  for(int i = 1; i <= n; ++i)
    if(!vis[i])
      isBiPartite &= checkBiPartite(i);
  cout << (isBiPartite ? "YES" : "NO") << endl;
}
```

## 2.9 Bellman ford

```
// 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.

bool hasNC() {
  for(int i = 1; i <= n; ++i)
    for(int j = Head[i]; j; j = Next[j])
      if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]])
  return true;

  return false;
}

bool Bellman_Ford(int src)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  dis[src] = 0;
  bool newRelaxation = true;

  for(int i = 2; i <= n && newRelaxation; ++i) {
    newRelaxation = false;
    for(int i = 1; i <= n; ++i)
      for(int j = Head[i]; j; j = Next[j])
  if(dis[i] < INF && dis[i] + Cost[j] < dis[To[j]]) {
    dis[To[j]] = dis[i] + Cost[j];
    Par[To[j]] = i;
    newRelaxation = true;
  }
  }
  return hasNC();
}
```

## 2.10 Connected components

```
void DFS(int node) {
  visited[node] = true;
  for(int e = Head[node]; e; e = Next[e])
    if(!visited[To[e]])
      DFS(To[e]);
}

int main() {
  for(int node = 1; node <= n; ++node)
    if(!visited[node])
      ++CCs, DFS(node);
  cout << CCs << endl;
}
```

## 2.11 Cycle detection

```
void DFS(int node, int parent = -1)
{
  if(hasCycle |= visited[node])
    return;
  visited[node] = true;

  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != parent)
      DFS(To[i], node);
}

int main() {
  for(int i = 1; i <= n; ++i)
    if(!visited[i])
      DFS(i);
  cout << (hasCycle ? "YES" : "NO") << endl;
}
```

## 2.12 Directed cyclic graph into acyclic

```
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;
bool in_stack[N];

void addEdge(int from, int to, int cost = 0) {
  Next[++ne] = Head[from];
  Head[from] = ne;
  Cost[ne] = cost;
  To[ne] = to;
}

void addEdgeDAG(int from, int to, int cost = 0) {
  NextDAG[++neDAG] = HeadDAG[from];
  HeadDAG[from] = neDAG;
  CostDAG[ne] = cost;
  ToDAG[neDAG] = to;
  ++out[from];
}

void _clear() {
  memset(Head,     0, sizeof(Head[0])     * (n + 2));
  memset(dfs_num,  0, sizeof(dfs_num[0])  * (n + 2));
  memset(compID,   0, sizeof(compID[0])   * (n + 2));
  memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
  memset(HeadDAG,  0, sizeof(HeadDAG[0])  * (n + 2));
  memset(out,      0, sizeof(out[0])      * (n + 2));
  ne = dfs_timer = top = neDAG = ID = 0;
}

void Tarjan(int node)
{
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  in_stack[Stack[top++] = node] = true;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0)
      Tarjan(To[i]);

    if(in_stack[To[i]])
      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
  }

  if(dfs_num[node] == dfs_low[node]) {
    ++ID;
    for(int cur = -1; cur ^ node;) {
      in_stack[cur = Stack[--top]] = false;
      compID[cur] = ID;
      ++compSize[ID];
    }
  }
}

void Tarjan() {
  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0)
      Tarjan(i);
}

void DFS(int node)
{
  dfs_num[node] = 1;
  for(int i = Head[node]; i; i = Next[i]) {
      if(compID[node] != compID[To[i]])
  addEdgeDAG(compID[node], compID[To[i]]);
      if(dfs_num[To[i]] == 0)
  DFS(To[i]);
    }
}

void construct_dag() {
  memset(dfs_num, 0, sizeof(dfs_num[0]) * (n + 2));

  for(int i = 1; i <= n; ++i)
    if(dfs_num[i] == 0)
      DFS(i);
}
```

## 2.13 Dijkstra (dense graph)

```
/** Dijkstra on dense graphs
    complexity : O(n^2 + m)
**/
int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void Dijkstra(int src, int V)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  vector <bool> mark(V + 1, false);

  dis[src] = 0;
  for(int i = 1; i <= V; ++i) {
    int node = 0;
    for(int j = 1; j <= V; ++j)
      if(!mark[j] && dis[j] < dis[node])
  node = j;

    if(dis[node] == INF) break;
    mark[node] = true;
    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]])
  {
    dis[To[i]] = dis[node] + Cost[i];
    Par[To[i]] = node;
  }
  }
}
```

## 2.14 Dijkstra (grid)

```
const int dr[]   = { 1, -1, 0, 0, 1, 1, -1, -1 };
const int dc[]   = { 0, 0, 1, -1, 1, -1, 1, -1 };
const char dir[] = {'D', 'U', 'R', 'L'};

const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;

int grid[N][N], dis[N][N], n, m;

bool valid(int r, int c) {
  return r >= 1 && r <= n && c >= 1 && c <= m;
}

void Dijkstra(int sr, int sc)
{
  memset(dis, 0x3f, sizeof (dis)); // memset(dis, 0x3f, n * m) we don't do that here

  priority_queue <tuple <int, int, int> > Q;
  dis[sr][sc] = grid[sr][sc];
  Q.push({-grid[sr][sc], sr, sc});

  int cost, r, c, nr, nc;
  while(Q.size())
    {
      tie(cost, r, c) = Q.top(); Q.pop();
      if((-cost) > dis[r][c]) continue; // lazy deletion

      for(int i = 0; i < 4; ++i)
        {
    nr = r + dr[i];
    nc = c + dc[i];

    if(!valid(nr, nc)) continue;

    if(dis[r][c] + grid[nr][nc] < dis[nr][nc])
            {
        dis[nr][nc] = dis[r][c] + grid[nr][nc];
        Q.push({-dis[nr][nc], nr, nc});
            }
        }
    }
}
```

## 2.15 Dijkstra (negative weighted graph)

```
int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void Dijkstra(int src)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  priority_queue <pair <ll, int> > Q;
  dis[src] = 0;
  Q.push({-dis[src], src});

  int node;
  ll cost;
  while(Q.size()) {
    tie(cost, node) = Q.top(); Q.pop();
    if((-cost) > dis[node]) continue;

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]])
  {
    dis[To[i]] = dis[node] + Cost[i];
    Q.push({-dis[To[i]], To[i]});
    Par[To[i]] = node;
  }
  }
}
```

## 2.16 Dijkstra (sparse graph)

```
/** Dijkstra on sparse graphs
    - complexity : O(n + m)logn -> O(nlogn + m)
    - Single Source Single Destination Shortest Path Problem
    - Positive Weight Edges only
    Subpaths of shortest paths from u to v are shortest paths!
**/
int Head[N], Par[N], Next[M], To[M], Cost[M], ne, n, m, u, v, st, tr, tax;
ll dis[N];

void Dijkstra(int src, int trg)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  priority_queue <pair <ll, int> > Q;
  dis[src] = 0;
  Q.push({-dis[src], src});

  int node;
  ll cost;
  while(Q.size()) {
    tie(cost, node) = Q.top(); Q.pop();

    if((-cost) > dis[node]) continue; // lazy deletion
    if(node == trg) return;           // cheapest cost in case of positive weight edges

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]])
  {
    dis[To[i]] = dis[node] + Cost[i];
    Q.push({-dis[To[i]], To[i]});
    Par[To[i]] = node;
  }
  }
}
```

## 2.17 Edge classification

```
int Head[N], Next[M], To[M], Par[N], in_time[N], ne, n, m, u, v, dfs_timer;
char dfs_num[N];

void edgeClassification(int node)
{
  dfs_num[node] = EXPLORED;
  in_time[node] = ++dfs_timer;

  for(int i = Head[node]; i; i = Next[i])
    {
      if(dfs_num[To[i]] == UNVISITED)
        {
    cout << "Tree Edge : " << node << " -> " << To[i] << endl;

    Par[To[i]] = node;
```

```
    edgeClassification(To[i]);
        }
      else if(dfs_num[To[i]] == VISITED)
        {
    /** Cross Edges only occur in directed graph */
    if(in_time[To[i]] < in_time[node])
      cout << "Cross Edge : " << node << " -> " << To[i] << endl;
    else
      cout << "Forward Edge : " << node << " -> " << To[i] << endl;
        }
      else if(dfs_num[To[i]] == EXPLORED)
        {
    if(Par[node] == To[i])
      cout << "Bi-Directional Edge : " << node << " -> " << To[i] << endl;
    else
      cout << "Backward Edge : " << node << " -> " << To[i] << " (Cycle)" << endl;
        }
    }

  dfs_num[node] = VISITED;
}

int main() {
  for(int i = 1; i <= n; ++i) if(!dfs_num[i])
        edgeClassification(i);
}
```

## 2.18 Eulerian tour tree

```
int Head[N], To[M], Next[M], Cost[M], ne, n, m, u, v, w, Last[N], First[N], euler_tour[1 + N << 1];
ll  Height[1 + N << 1];
int euler_timer;

void _clear() {
  memset(Head,       0, sizeof(Head[0])       * (n + 2));
  memset(Last,       0, sizeof(Last[0])       * (n + 2));
  memset(First,      0, sizeof(First[0])      * (n + 2));
  ne = euler_timer = 0;
}

/**
   euler_tour[1 .. n * 2 - 1] = which records the sequence of visited nodes
   Height[1 .. n * 2 - 1]     = which records the depth of each visited node

   First[1 .. n]      = records the index of the first occurrence of node i in euler_tour
   Last[1 .. n]       = records the index of the last  occurrence of node i in euler_tour
**/

void EulerianTour(int node, ll depth = 0)
{
  euler_tour[++euler_timer] = node;
  Height[euler_timer] = depth;
  First[node] = euler_timer;

  for(int i = Head[node]; i; i = Next[i])
    if(First[To[i]] == 0)
      {
  EulerianTour(To[i], depth + Cost[i]);

  euler_tour[++euler_timer] = node;
  Height[euler_timer] = depth;
      }

  Last[node] = euler_timer;
}

void show() {
  for(int i = 1; i < (n << 1); ++i) cout << euler_tour[i] << " ";cout << endl;
  for(int i = 1; i < (n << 1); ++i) cout << Height[i] << " ";    cout << endl;
  for(int i = 1; i <= n; ++i)       cout << First[i] << " ";     cout << endl;
  for(int i = 1; i <= n; ++i)       cout << Last[i] << " ";      cout << endl;
}

int main()
{
  EulerianTour(1);
  show();
}
```

## 2.19 Floodfill

```
/** check if there is a path from (0, 0) to (n - 1, m - 1) using '.' only **/

int dr[4] = {1, -1, 0, 0};
int dc[4] = {0, 0, 1, -1};
char grid[N][M];
int n, m;

bool valid(int r, int c) {
  return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] == '.';
}

bool isDis(int r, int c) {
  return r == n - 1 && c == m - 1;
}

bool FloodFill(int r, int c) {
  if(!valid(r, c)) return false;
  if(isDis(r, c)) return true;

  grid[r][c] = '#';
  for(int i = 0; i < 4; ++i)
    if(FloodFill(r + dr[i], c + dc[i])) return true;

  return false;
}

int main() {
  cout << (FloodFill(0, 0) ? "YES" : "NO") << endl;
}
```

## 2.20 Floyd warshall (all-pairs shortest path)

```
/** -The graph has a 'negative cycle' if at the end of the algorithm,
  the distance from a vertex v to itself is negative.

  - before k-th phase the value of d[i][j] is equal to the length of
  the shortest path from vertex i to the vertex j,
  if this path is allowed to enter only the vertex with numbers smaller than k
  (the beginning and end of the path are not restricted by this property).
**/

int Par[N][N], n, m, u, v, tax;
ll adj[N][N], dis[N][N];

vector <int> restorePath(int st, int tr)
{
  vector <int> path;
  if(dis[st][tr] == INF) return path;

  for(int i = tr; st ^ i; i = Par[st][i])
    path.push_back(i);

  path.push_back(st);
  reverse(path.begin(), path.end());
  return path;
}

void Floyd_Warshall()
{
  for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= n; ++j)
      Par[i][j] = i;

  for(int k = 1; k <= n; ++k)
    for(int i = 1; i <= n; ++i)
      for(int j = 1; j <= n; ++j)
  if(dis[i][k] + dis[k][j] < dis[i][j])
    {
      dis[i][j] = dis[i][k] + dis[k][j];
      Par[i][j] = Par[k][j];
    }
}
```

## 2.21 Minimum spanning tree (Kruskal)

```
int n, m, u, v, w;
vector < tuple <int, int, int> > edges;
UnionFind uf;

pair < ll, vector < pair <int, int> > > Kruskal()
{
  sort(edges.begin(), edges.end());

```

```
  vector < pair <int, int> > mstEdges;
  int from, to, cost;
  ll minWieght = 0;

  for(tuple <int, int, int> edge : edges)
    {
      tie(cost, from, to) = edge;
      if(uf.union_set(from, to))
        {
    minWieght += cost;
    mstEdges.push_back(make_pair(from, to));
        }
    }

  if(mstEdges.size() == n - 1)
    return make_pair(minWieght, mstEdges);

  return make_pair(-1, vector < pair <int, int> > ());
}
```

## 2.22 Kth ancestor and lowest common ancestor (binary lifting)

```
int Head[N], To[M], Next[M], Par[N], up[N][LOG + 1], Log[N], Level[N], ne, n, u, v, q;

void _clear() {
  memset(Head,  0, sizeof(Head[0])   * (n + 2));
  memset(Par,   0, sizeof(Par[0])    * (n + 2));
  memset(Level, 0, sizeof(Level[0])  * (n + 2));
  ne = 0;
}

int lastBit(int a) {
  return (a & -a);
}

void logCalc()
{
  Log[1] = 0;
  for(int i = 2; i < N; ++i)
    Log[i] = Log[i >> 1] + 1;
}

void DFS(int node, int depth = 0)
{
  Level[node] = depth;
  up[node][0] = Par[node];  // Par[root] = root

  for(int i = 1; i <= LOG; ++i) {
    up[node][i] = up[up[node][i - 1]][i - 1];
  }

  for(int i = Head[node]; i; i = Next[i]) if(To[i] != Par[node]) {
      Par[To[i]] = node;
      DFS(To[i], depth + 1);
    }
}

int KthAncestor(int u, int k)
{
  if(k > Level[u]) return -1;

  for(int i = lastBit(k); k; k -= lastBit(k), i = lastBit(k))
    u = up[u][Log[i]];

  return u;
}

int LCA(int u, int v)
{
  if(Level[u] < Level[v]) swap(u, v);
  int k = Level[u] - Level[v];

  u = KthAncestor(u, k);
  if(u == v) return u;

  for(int i = LOG; i >= 0; --i)
    if(up[u][i] ^ up[v][i])
      {
  u = up[u][i];
  v = up[v][i];
      }

  return up[u][0];
}

int main()
{
  logCalc();
  for(int i = 1; i <= n; ++i) if(Par[i] == 0) {
      Par[i] = i;
      DFS(i);
    }

  cin >> q;
  while(q--)
    {
      cin >> u >> v;
      cout << LCA(u, v) << endl;
    }
}
```

## 2.23 Lowest common ancestor (euler tour)

```
int Head[N], To[M], Next[M], Cost[M], ne, n, m, u, v, w, q;
int Last[N], First[N], euler_tour[N << 1], Height[N << 1], euler_timer;

void EulerianTour(int node, int depth = 0)
{
  euler_tour[++euler_timer] = node;
  Height[euler_timer] = depth;
  First[node] = euler_timer;

  for(int i = Head[node]; i; i = Next[i])
    if(First[To[i]] == 0)
      {
  EulerianTour(To[i], depth + Cost[i]);

  euler_tour[++euler_timer] = node;
  Height[euler_timer] = depth;
      }

  Last[node] = euler_timer;
}

int main()
{
  EulerianTour(1);
  SparseTable <int> st(Height + 1, Height + euler_timer + 1, [&](int a, int b) { return a <= b; });

  int l, r; cin >> q;
  while(q--)
    {
      cin >> l >> r;

      int left = Last[l];
      int right = Last[r];
      if(left > right) swap(left, right);

      cout << euler_tour[ st.query(left, right) ] << endl;
    }
}
```

## 2.24 Minimum vertex cover

```
bool DFS(int node, int par = -1) {

  bool black = false;
  for(int e = Head[node]; e; e = Next[e])
    if(To[e] != par)
      black |= DFS(To[e], node);

  MVC += black;
  return !black;
}
```

## 2.25 Shortest path faster algorithml (SPFA)

```
/** Shortest Path Faster Algorithm :
    - This algorithm runs in O(kE) where k is a number depending on the graph.
    - The maximum k can be V (which is the same as the time complexity of Bellman Fords).
    - However in practice SPFA (which uses a queue) is as fast as Dijkstras (which uses a priority
         queue).
    - SPFA can deal with negative weight edge. If the graph has no negative cycle, SPFA runs well on
         it.
```

```
    - If the graph has negative cycle(s), SPFA can also detect it as there must be some vertex (those
         on the negative cycle)
    that enters the queue for over V dash 1 times.
**/

int Head[N], Par[N], Next[M], To[M], Cost[M], Cnt[N], ne, n, m, u, v, st, tax;
ll dis[N];
bool Inq[N];

void _set() {
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));
  memset(Cnt,    0, sizeof(Cnt[0]) * (n + 2));
  memset(Inq,    0, sizeof(Inq[0]) * (n + 2));
}

bool SPFA(int src)
{
  _set();

  deque <int> Q;
  Q.push_front(src);

  dis[src] = 0;
  Cnt[src] = 1;
  Inq[src] = 1;

  int node;
  while(Q.size()) {
    node = Q.front(); Q.pop_front(); Inq[node] = 0;

    for(int i = Head[node]; i; i = Next[i])
      if(dis[node] + Cost[i] < dis[To[i]]) {
  dis[To[i]] = dis[node] + Cost[i];
  Par[To[i]] = node;

  if(!Inq[To[i]])
    {
      if(++Cnt[To[i]] == n)
        return true; // graph has a negative weight cycle

      if(Q.size() && dis[To[i]] > dis[Q.front()])
        Q.push_back(To[i]);
      else
        Q.push_front(To[i]);

      Inq[To[i]] = true;
    }
      }
  }
  return false;
}
```

## 2.26 Single source shortest path

```
int Head[N], Par[N], Next[M], To[M], ne, n, m, u, v, st, tr;
ll dis[N];

void BFS(int src)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  queue <int> Q;
  Q.push(src);
  dis[src] = 0;

  int node;
  while(Q.size()) {
    node = Q.front(); Q.pop();
    for(int i = Head[node]; i; i = Next[i])
      if(dis[To[i]] == INF) {
  dis[To[i]] = dis[node] + 1;
  Par[To[i]] = node;
  Q.push(To[i]);
      }
  }
}
```

## 2.27 Single source shortest path (grid)

```
const int dr []     = {-1, 0, 1, 0};
const int dc []     = {0, 1, 0, -1};
const char dir []   = {'U', 'R', 'D', 'L'};
map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};

int dis[N][N], n, m;
char Par[N][N];

bool valid(int r, int c) {
  return r >= 1 && r <= n && c >= 1 && c <= m && dis[r][c] == oo;
}

void BFS(int sr, int sc)
{
  memset(dis, 0x3f, sizeof(dis));
  memset(Par,   -1, sizeof(Par));

  queue < pair <int, int> > Q;
  dis[sr][sc] = 0;
  Q.push({sr, sc});

  int r, c, nr, nc;
  while(Q.size()) {
    tie(r, c) = Q.front(); Q.pop();

    for(int i = 0; i < 4; ++i) {
      nr = r + dr[i];
      nc = c + dc[i];

      if(!valid(nr, nc)) continue;
      dis[nr][nc] = dis[r][c] + 1;
      Par[nr][nc] = dir[i ^ 2];
      Q.push({nr, nc});
    }
  }
}
```

## 2.28 Tarjan (strongly connected components)

```
int Head[N], To[M], Next[M], Cost[M];
int dfs_num[N], dfs_low[N];
int Stack[N], compID[N], compSize[N];
int ne, n, m, u, v, w;
int dfs_timer, top, ID;
bool in_stack[N];

void _clear() {
  memset(Head,     0, sizeof(Head[0])     * (n + 2));
  memset(dfs_num,  0, sizeof(dfs_num[0])  * (n + 2));
  memset(compID,   0, sizeof(compID[0])   * (n + 2));
  memset(compSize, 0, sizeof(compSize[0]) * (n + 2));
  ne = dfs_timer = top = ID = 0;
}

void Tarjan(int node)
{
  dfs_num[node] = dfs_low[node] = ++dfs_timer;
  in_stack[Stack[top++] = node] = true;

  for(int i = Head[node]; i; i = Next[i]) {
    if(dfs_num[To[i]] == 0)
      Tarjan(To[i]);

    if(in_stack[To[i]])
      dfs_low[node] = Min(dfs_low[node], dfs_low[To[i]]);
  }

  if(dfs_num[node] == dfs_low[node]) {
    ++ID;
    for(int cur = -1; cur ^ node;) {
      in_stack[cur = Stack[--top]] = false;
      compID[cur] = ID;
      ++compSize[ID];
    }
  }
}

void Tarjan() {
  for(int i = 1; i <= n; ++i) if(dfs_num[i] == 0)
        Tarjan(i);
}
```

## 2.29 Tree diameter

```
void DFS(int node, int cost, int par = -1) {
  if(cost > diameter) diameter = cost, at = node;
  for (int e = Head[node]; e; e = Next[e]) if(To[e] != par)
              DFS(To[e], cost + 1, node);
}

int main() {
  DFS(1, 0);
  from = at, diameter = 0;
  DFS(from, 0);
  to = at;
  cout << diameter << " " << from << " " << to << endl;
}
```

## 2.30 Tree diameter (weighted tree)

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

int main() {
  DFS(1, 0ll);
  from = at, diameter = 0;
  DFS(from, 0ll);
  to = at;
  cout << diameter << endl;
}
```

## 2.31 Topological sort (kahns algorithm)

```
vector <int> kahn(int n)
{
  vector <int> ready, ret;

  for(int i = 1; i <= n; ++i)
    if(!in[i])
      ready.push_back(i);

  int node;
  while(!ready.empty())
    {
      node = ready.back(); ready.pop_back();
      ret.push_back(node);

      for(int i = Head[node]; i; i = Next[i])
  if(--in[To[i]] == 0)
    ready.push_back(To[i]);
    }
  return ret;
}

int main() {
  vector <int> v = kahn(n);
  if((int)v.size() == n)
    for(int i : v)
      cout << i << ' ';
  else
    cout << "not a DAG!" << endl;
}
```

## 2.32 Problems

### 2.32.1 Restoring the path

```
const int dr []     = {-1, 0, 1, 0};
const int dc []     = {0, 1, 0, -1};
const char dir []   = {'U', 'R', 'D', 'L'};
map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};

/** Implicit Graphs

    - in BFS, Dijkstra or Bellman-Ford function write -> Par[nr][nc] = dir[i ^ 2]
    - char Par[N][N] initialize with -1
    - si start i
    - sj strat j
    - fi target i
    - fj target j
    - char dir and its map inv
    - dr, dc
**/

string restorePath(int si, int sj, int fi, int fj) {
  string s;
  if(Par[ei][ej] == -1) return s;

  int ei = fi, ej = fj;
  for(char i = Par[fi][fj]; (si ^ fi) || (sj ^ fj); i = Par[fi][fj]) {
    s  += dir[inv[i] ^ 2];
    fi += dr[inv[i]];
    fj += dc[inv[i]];
  }

  reverse(s.begin(), s.end());
  return s;
}

/** Explicit Graphs (BFS, Dijkstra or Bellman-Ford)

    - int Par[N] initialize with  -1
    - ll dis[N]  initialize with  0x3f
    - ll INF = 0x3f3f3f3f3f3f3f3f
**/

vector <int> restorePath(int dest) {
  vector <int> path;
  if(dis[dest] == INF) return path;

  for(int i = dest; ~i; i = Par[i])
    path.push_back(i);

  reverse(path.begin(), path.end());
  return path;
}

/** in case of Floyd-Warshall:

    - ll dis[N][N] initialize with 0x3f
    - ll INF = 0x3f3f3f3f3f3f3f3f
    - int Par[N][N] initialize with       Par[i][j] = i;
    - in Floyd-Warshall function write -> Par[i][j] = Par[k][j];
**/

vector <int> restorePath(int st, int tr) {
  vector <int> path;
  if(dis[st][tr] == INF) return path;

  for(int i = tr; st ^ i; i = Par[st][i])
    path.push_back(i);

  path.push_back(st);
  reverse(path.begin(), path.end());
  return path;
}
```

### 2.32.2 Tree distances

```
int Head[N], Next[M], To[M], Par[N], ne, n, m, u, v, diameter, At, From;

int E[N << 1], H[N << 1], F[N], L[N], timer, SP[N << 1][LOG + 1], Log[N << 1];

void _clear() {
  memset(Head, 0, sizeof(Head[0]) * (n + 2));
  memset(Par, -1, sizeof(Par[0])  * (n + 2));
  ne = 0;
  timer = 0;
}

void EulerTour(int node, int depth = 0, int par = -1) {
  E[++timer] = node;
  H[timer] = depth;
  F[node] = timer;

  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par) {
      EulerTour(To[i], depth + 1, node);
      E[++timer] = node;
      H[timer] = depth;
    }

  L[node] = timer;
```

```
}

void dfs(int node, int depth = 0, int par = -1) {
  if(depth > diameter) diameter = depth, At = node;
  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par)
      dfs(To[i], depth + 1, node);
}

void bulid()
{
  EulerTour(1);
  dfs(1); From = At; diameter = 0; dfs(From);

  Log[1] = 0;
  for(int i = 2; i <= (n << 1); ++i)
    Log[i] = Log[i >> 1] + 1;

  for(int i = 1; i < (n << 1); ++i)
    SP[i][0] = i;

  int MaxLog = Log[(n << 1)];
  for(int j = 1, k, h; j <= MaxLog; ++j) {
    k = (1 << j);
    h = (k >> 1);
    for(int i = 1; i + k - 1 < (n << 1); ++i)
      {
  const int & x = SP[i][j - 1];
  const int & y = SP[i + h][j - 1];

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

int query(int l, int r)
{
  int d = r - l + 1;
  int lg = Log[d];
  int k = (1 << lg);

  const int & x = SP[l][lg];
  const int & y = SP[l + d - k][lg];

  return (H[x] <= H[y] ? x : y);
}

int LCA(int u, int v) {
  return query(u, v);
}

int distance(int u, int v) {
  int l = F[u];
  int r = F[v];
  if(l > r) swap(l, r);

  int ix = LCA(l, r);
  return (H[l] + H[r] - H[ix] - H[ix]);
}

int main()
{
  bulid();
  for(int i = 1; i <= n; ++i)
    cout << max(distance(i, At), distance(i, From)) << " \n"[i == n];
}
```

### 2.32.3 Subtree sizes

```
int Head[N], Next[M], To[M], Par[N], sbtree_size[N], ne, n, m, u, v, w;

void dfs(int node, int par = -1) {
  sbtree_size[node] = 1;
  for(int i = Head[node]; i; i = Next[i])
    if(To[i] != par) {
      dfs(To[i], node);
      sbtree_size[node] += sbtree_size[To[i]];
    }
}

int main()
{
  dfs(1);
  for(int i = 1; i <= n; ++i)
    cout << sbtree_size[i] - 1 << " \n"[i == n];
}
```

### 2.32.4 Shortest cycle

```
/** for each node run BFS and minmize the cycle length
 **/

int BFS(int src)
{
  memset(dis, 0x3f, sizeof(dis[0]) * (n + 2));
  memset(Par,   -1, sizeof(Par[0]) * (n + 2));

  queue <int> Q;
  Q.push(src);
  dis[src] = 0;

  int node, ret = oo;
  while(Q.size())
    {
      node = Q.front(); Q.pop();
      for(int i = Head[node]; i; i = Next[i])
        {
    if(dis[To[i]] != oo) {
      if(Par[node] != To[i]) {
        if(dis[node] + 1 + dis[To[i]] < ret)
    ret = dis[node] + 1 + dis[To[i]];
      }
      continue;
    }

    dis[To[i]] = dis[node] + 1;
    Par[To[i]] = node;
    Q.push(To[i]);
        }
    }
  return ret;
}
```

### 2.32.5 Path sum for each node

```
int Head[N], Next[M], To[M], ne, u, v, n, m, subtree_size[N], level[N];
ll dis[N];

void dfs(int node, int par = -1) {
  subtree_size[node] = 1;
  for(int i = Head[node]; i; i = Next[i]) if(To[i] != par) {
      level[To[i]] = level[node] + 1;
      dfs(To[i], node);
      subtree_size[node] += subtree_size[To[i]];
    }
}

void reRoot(int node, ll pd, int par = -1) {
  dis[node] = pd;
  for(int i = Head[node]; i; i = Next[i]) if(To[i] != par) {
      reRoot(To[i], pd - subtree_size[To[i]] + (n - subtree_size[To[i]]), node);
    }
}

void get_dis()
{
  dfs(1);
  ll pd = 0;
  for(int i = 1; i <= n; ++i)
    pd += level[i];

  reRoot(1, pd);
  for(int i = 1; i <= n; ++i)
    cout << dis[i] << " \n"[i == n];
}
```

# 3 Data structures

## 3.1 Segment tree

```
template <class T, class F = function <T(const T &, const T &)> >
class SegmentTree
{
  vector <T> _A;
```

```
  vector <T> ST;
  vector <T> LT;
  F func;
  int _N;

public :
  template <class iter>
  SegmentTree(iter _begin, iter _end, const F _func = [](T a, T b) {return a <= b ? a : b;}) : func(_func)
  {
    _N = distance(_begin, _end);
    _N = (1 << (int)ceil(log2(_N)));

    _A.assign(_N + 1,  0);
    ST.assign(_N << 1, 0);
    LT.assign(_N << 1, 0);

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; ++i, ++j)
      _A[j] = *i;

    build(1, 1, _N);
  }

  void build(int p, int l, int r)
  {
    if(l == r) {
      ST[p] = _A[l];
      return;
    }

    int mid = (l + r) >> 1;

    build(p + p, l, mid);
    build(p + p + 1, mid + 1, r);

    const T & x = ST[p + p];
    const T & y = ST[p + p + 1];

    ST[p] = func(x, y);
  }

  void update_range(int ul, int ur, int delta) {
    update_range(ul, ur, delta, 1, 1, _N);
  }

  T query(int ql, int qr) {
    return query(ql, qr, 1, 1, _N);
  }

  void update_point(int inx, int delta)
  {
    inx += _N - 1;
    ST[inx] = delta;

    while(inx > 1) {
      inx >>= 1;

      const T & x = ST[inx + inx];
      const T & y = ST[inx + inx + 1];

      ST[inx] = func(x, y);
    }
  }

private :
  void update_range(int ul, int ur, int delta, int p, int l, int r)
  {
    if(r < ul || ur < l)
      return;

    if(ul <= l && r <= ur) {
      ST[p] += delta;
      LT[p] += delta;
      return;
    }

    propagate(p);

    int mid = (l + r) >> 1;

    update_range(ul, ur, delta, p + p, l, mid);
    update_range(ul, ur, delta, p + p + 1, mid + 1, r);

    const T & x = ST[p + p];
    const T & y = ST[p + p + 1];

    ST[p] = func(x, y);
  }

  T query(int ql, int qr, int p, int l, int r)
  {
    if(r < ql || qr < l)
      return INT_MAX;

    if(ql <= l && r <= qr)
      return ST[p];

    propagate(p);

    int mid = (l + r) >> 1;

    const T & x = query(ql, qr, p + p, l, mid);
    const T & y = query(ql, qr, p + p + 1, mid + 1, r);

    return  func(x, y);
  }

  void propagate(int p) {
    if(LT[p]) {
      ST[p + p]     += LT[p];
      ST[p + p + 1] += LT[p];
      LT[p + p]     += LT[p];
      LT[p + p + 1] += LT[p];
      LT[p] = 0;
    }
  }
};
```

## 3.2 Merge sort tree

```
class SegmentTree
{
  vector <vector <int> > sTree;
  vector <int> localArr;
  int NP2, oo = 0x3f3f3f3f;

public :
  template <class T>
  SegmentTree(T _begin, T _end)
  {
    NP2 = 1;
    int n = _end - _begin;
    while(NP2 < n) NP2 <<= 1;

    sTree.assign(NP2 << 1, vector <int> ());
    localArr.assign(NP2 + 1, 0);

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; i++, ++j)
      localArr[j] = *i;

    build(1, 1, NP2);
  }

  void build(int p, int l, int r)
  {
    if(l == r) {
      sTree[p].push_back(localArr[l]);
      return;
    }

    build(left(p),  l,     mid(l, r));
    build(right(p), mid(l, r) + 1, r);

    merge(p);
  }

  int query(int ql, int qr, int k) {
    return query(ql, qr, k, 1, 1, NP2);
  }

private :
  int query(int ql, int qr, int k, int p, int l, int r)
  {
    if(isOutside(ql, qr, l, r))
      return 0;

    if(isInside(ql, qr, l, r)) {
      return sTree[p].end() - upper_bound(sTree[p].begin(), sTree[p].end(), k);
    }

    return query(ql, qr, k, left(p),  l,     mid(l, r)) +
      query(ql, qr, k, right(p), mid(l, r) + 1, r);
  }

  void merge(int p)
  {
    vector <int> & L = sTree[left(p)];
```

```
    vector <int> & R = sTree[right(p)];

    int l_size = L.size();
    int r_size = R.size();
    int p_size = l_size + r_size;

    L.push_back(INT_MAX);
    R.push_back(INT_MAX);

    sTree[p].resize(p_size);

    for(int k = 0, i = 0, j = 0; k < p_size; ++k)
      if(L[i] <= R[j])
  sTree[p][k] = L[i], i += (L[i] != INT_MAX);
      else
  sTree[p][k] = R[j], j += (R[j] != INT_MAX);

    L.pop_back();
    R.pop_back();
  }

  inline bool isInside(int ql, int qr, int sl, int sr) {
    return (ql <= sl && sr <= qr);
  }

  inline bool isOutside(int ql, int qr, int sl, int sr) {
    return (sr < ql || qr < sl);
  }

  inline int mid (int l, int r) {
    return ((l + r) >> 1);
  }

  inline int left(int p) {
    return (p << 1);
  }

  inline int right(int p) {
    return ((p << 1) | 1);
  }
};
```

## 3.3 Merge sort

```
ll inversions;

template <class T>
void merge(T localArr [], int l, int mid, int r)
{
  int l_size = mid - l + 1;
  int r_size = r - mid;

  T L[l_size + 1];
  T R[r_size + 1];

  for(int i = 0; i < l_size; ++i) L[i] = localArr[i + l];
  for(int i = 0; i < r_size; ++i) R[i] = localArr[i + mid + 1];

  T Mx;
  if(sizeof(T) == 4) Mx = INT_MAX;
  else Mx = LONG_MAX;

  L[l_size] = R[r_size] = Mx;

  for(int k = l, i = 0, j = 0; k <= r; ++k)
    if(L[i] <= R[j])
      localArr[k] = L[i], i += (L[i] != Mx);
    else
      localArr[k] = R[j], j += (R[j] != Mx), inversions += l_size - i;
}

template <class T>
void merge_sort(T localArr [], int l, int r)
{
  if(r - l)
    {
      int mid = (l + r) >> 1;
      merge_sort(localArr, l,     mid);
      merge_sort(localArr, mid + 1, r);
      merge(localArr,      l,  mid, r);
    }
}

template <class T>
void merge_sort(T _begin, T _end)
{
  const int sz = _end - _begin;
  __typeof(*_begin) localArray[sz];

  __typeof(_begin) k = _begin;
  for(int i = 0; k != _end; ++i, ++k)
    localArray[i] = *k;

  merge_sort(localArray, 0, sz - 1);

  k = _begin;
  for(int i = 0; k != _end; ++i, ++k)
    *k = localArray[i];
}
```

## 3.4 Sparse table

```
template <class T, class F = function <T(const T&, const T&)> >
class SparseTable
{
  int _N;
  int _LOG;
  vector <T> _A;
  vector < vector <T> > ST;
  vector <int> Log;
  F func;

public :
  SparseTable() = default;

  template <class iter>
  SparseTable(iter _begin, iter _end, const F _func = less <T> ()) : func(_func)
  {
    _N = distance(_begin, _end);

    Log.assign(_N + 1, 0);
    for(int i = 2; i <= _N; ++i)
      Log[i] = Log[i >> 1] + 1;

    _LOG = Log[_N];

    _A.assign(_N + 1, 0);
    ST.assign(_N + 1, vector <T> (_LOG + 1, 0));

    __typeof(_begin) i = _begin;
    for(int j = 1; i != _end; ++i, ++j)
      _A[j] = *i;

    build();
  }

  void build()
  {
    for(int i = 1; i <= _N; ++i)
      ST[i][0] = i;

    for(int j = 1, k, d; j <= _LOG; ++j)    // the two nested loops below have overall time complexity = O(n log n)
      {
  k = (1 << j);
  d = (k >> 1);

  for(int i = 1; i + k - 1 <= _N; ++i)
    {
      T const & x = ST[i][j - 1];     // starting subarray at index = i     with length = 2^{j - 1}
      T const & y = ST[i + d][j - 1]; // starting subarray at index = i + d with length = 2^{j - 1}

      ST[i][j] = func(_A[x], _A[y]) ? x : y;
    }
      }
  }

  T query(int l, int r) // this query is O(1)
  {
    int d = r - l + 1;
    T const & x = ST[l][Log[d]];
    T const & y = ST[l + d - (1 << Log[d])][Log[d]];

    return func(_A[x], _A[y]) ? x : y;
  }
};
```

## 3.5 Union find disjoint sets

```cpp
/**
  Maintain a set of elements partitioned into non-overlapping subsets. Each
  partition is assigned a unique representative known as the parent, or root. The
  following implements two well-known optimizations known as union-by-size and
  path compression. This version is simplified to only work on integer elements.

   find_set(u) returns the unique representative of the partition containing u.
   same_set(u, v) returns whether elements u and v belong to the same partition.
   union_set(u, v) replaces the partitions containing u and v with a single new
  partition consisting of the union of elements in the original partitions.

  Time Complexity:
   O(a(n)) per call to find_set(), same_set(), and union_set(), where n is the
  number of elements, and a(n) is the extremely slow growing inverse of the Ackermann function
  (effectively a very small constant for all practical values of n).

  Space Complexity:
  O(n) for storage of the disjoint set forest elements.
   O(1) auxiliary for all operations.
**/

class UnionFind
{
  vector <int> par;
  vector <int> siz;
  int num_sets;
  size_t sz;

public:
  UnionFind() : par(1, -1), siz(1, 1), num_sets(0), sz(0) {}
  UnionFind(int n) : par(n + 1, -1), siz(n + 1, 1), num_sets(n), sz(n) {}

  int find_set(int u)
  {
    assert(u <= sz);

    int leader;
    for(leader = u; ~par[leader]; leader = par[leader]);

    for(int next = par[u]; u != leader; next = par[next]) {
      par[u] = leader;
      u = next;
    }
    return leader;
  }

  bool same_set(int u, int v) {
    return find_set(u) == find_set(v);
  }

  bool union_set(int u, int v) {
    if(same_set(u, v)) return false;

    int x = find_set(u);
    int y = find_set(v);

    if(siz[x] < siz[y]) swap(x, y);

    par[y] = x;
    siz[x] += siz[y];

    --num_sets;
    return true;
  }

  int number_of_sets() {
    return num_sets;
  }

  int size_of_set(int u) {
    return siz[find_set(u)];
  }

  size_t size() {
    return sz;
  }

  void clear() {
    par.clear();
    siz.clear();
    sz = num_sets = 0;
  }

  void assign(size_t n) {
    par.assign(n + 1, -1);
    siz.assign(n + 1,  1);
    sz = num_sets = n;
  }

  map < int, vector <int> > groups(int st) {
    map < int, vector <int> > ret;

    for(size_t i = st; i < sz + st; ++i)
      ret[find_set(i)].push_back(i);

    return ret;
  }
};
```

## 3.6 Bubble sort

```cpp
/**
  Bubble sort consists of n rounds. On each round, the algorithm iterates
  through the elements of the array. Whenever two consecutive elements are found
  that are not in correct order, the algorithm swaps them. The algorithm can be
  implemented as follows:
**/

template <class T>
void bubble_sort(T _begin, T _end, int round) {
  const int sz = _end - _begin;
  int localArray[sz];

  __typeof(_begin) k = _begin;
  for(int j = 0; k != _end; ++k, ++j)
    localArray[j] = *k;

  round = min(round, sz);
  for(int i = 0; i < round; ++i) /* n rounds -> n_th element **/
    for(int j = 0; j < sz - 1; ++j) if(localArray[j] > localArray[j + 1])
             swap(localArray[j], localArray[j + 1]);

  k = _begin;
  for(int j = 0; k != _end; ++k, ++j)
    *k = localArray[j];
}

/**
  After the first round of the algorithm, the largest element will be in the correct
  position, and in general, after k rounds, the k largest elements will be in the
  correct positions.
**/
```

# 4 Mathematics

## 4.1 Pisano periodic sequence

```cpp
template <class T>
using matrix = vector < vector <T> >;

template <class T> string to_string(T x) {
  int sn = 1;
  if(x < 0) sn = -1, x *= sn;
  string s = "";
  do {
    s = "0123456789"[x % 10] + s, x /= 10;
  } while(x);
  return (sn == -1 ? "-" : "") + s;
}

auto str_to_int(string x) {
  ui128 ret = (x[0] == '-' ? 0 : x[0] - '0');
  for(int i = 1; i < (int)x.size(); ++i) ret = ret * 10 + (x[i] - '0');
  return (x[0] == '-' ? -1 * (i128)ret : ret);
}

istream & operator >> (istream & in, i128 & i) noexcept {
  string s;
  in >> s;
  i = str_to_int(s);
  return in;
}

ostream & operator << (ostream & os, const i128 i) noexcept {
  os << to_string(i);
  return os;
}

void Fast() {
  cin.sync_with_stdio(0);
  cin.tie(0);
```

```cpp
  cout.tie(0);
}

ll n;
vector <int> primes;
matrix <ll> fibMatrix = {{1, 1},
       {1, 0}
};

i128 gcd(i128 a, i128 b) {
  while (a && b)
    a > b ? a %= b : b %= a;
  return a + b;
}

i128 lcm(i128 a, i128 b) {
  return a / gcd(a, b) * b;
}

vector < array <ll, 2> > factorize(ll x) {
  vector < array <ll, 2> > ret;
  for(int i = 0; 1ll * primes[i] * primes[i] <= x; ++i) {
    if(x % primes[i]) continue;

    int cnt = 0;
    while (x % primes[i] == 0) {
      cnt++;
      x /= primes[i];
    }
    ret.push_back({primes[i], cnt});
  }

  if(x > 1) ret.push_back({x, 1});
  return ret;
}

matrix <ll> MatMul(matrix <ll> A, matrix <ll> B, ll mod) {
  int ra = A.size(), cb = B[0].size(), ca = A[0].size();
  matrix <i128> C(ra, vector <i128> (cb));

  for(int i = 0; i < ra; ++i)
    for (int j = 0; j < cb; ++j) {
      C[i][j] = 0;
      for(int k = 0; k < ca; ++k)
  C[i][j] = (C[i][j] + (i128)A[i][k] * B[k][j]);
    }

  matrix <ll> ret(ra, vector <ll> (cb));
  for(int i = 0; i < ra; ++i)
    for (int j = 0; j < cb; ++j)
      ret[i][j] = C[i][j] % mod;

  return ret;
}

matrix <ll> MatPow(matrix <ll> A, ll p, ll mod) {
  int r = A.size(), c = A[0].size();
  assert(r == c && p);
  matrix <ll> result = A;
  p--;

  while(p) {
    if(p & 1ll) result = MatMul(result, A, mod);
    A = MatMul(A, A, mod);
    p >>= 1ll;
  }
  return result;
}

i128 ModExp(i128 a, ll p) {
  i128 result = 1;
  while(p) {
    if(p & 1ll) result = result * a;
    a *= a;
    p >>= 1ll;
  }
  return result;
}

ll nthFib(ll n, ll mod) {
  return MatPow(fibMatrix, n, mod)[0][1];
}

bool is_period(ll n, ll mod) {
  return nthFib(n, mod) == 0 && nthFib(n + 1, mod) == 1;
}

ll solver(ll x, ll mod) {
  vector < array <ll, 2> > factors = factorize(x);
  for(int i = 0; i < (int)factors.size(); ++i) {
    while(x % factors[i][0] == 0 && is_period(x / factors[i][0], mod))
      x /= factors[i][0];
  }
  return x;
}

ll pisano_prime(ll val) {
  if(val == 2) return 3;
  if(val == 5) return 20;
  if(val % 10 == 1 || val % 10 == 9)
    return solver(val - 1, val);

  return solver(2 * (val + 1), val);
}

const int N = 1e7 + 9;
bitset <N> isPrime;

void Precomputation_Sieve() {
  isPrime.set();
  int _sqrt = sqrtl(N);

  for(int i = 5; i <= _sqrt; i += 6) {
    if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
    i += 2;
    if(isPrime[i]) for (int j = i * i; j < N; j += i + i) isPrime.reset(j);
    i -= 2;
  }
}

vector <int> Primes(int n) {
  vector <int> _Primes;

  if(n >= 2) _Primes.push_back(2);
  if(n >= 3) _Primes.push_back(3);

  for (int i = 5; i <= n; i += 6) {
    if(isPrime[i]) _Primes.push_back(i);
    i += 2;
    if(isPrime[i]) _Primes.push_back(i);
    i -= 2;
  }
  return _Primes;
}

void initialize()
{
  Precomputation_Sieve();
  primes = Primes(N);
}

void Solve() {
  cin >> n;
  vector < array <ll, 2> > factors = factorize(n);

  i128 ans = 1;
  for (int i = 0; i < (int)factors.size(); ++i) {
    ans = lcm(ans, (i128)pisano_prime(factors[i][0]) * ModExp(factors[i][0], factors[i][1] - 1));
  }
  cout << ans << endl;
}

void MultiTest(bool Tests)
{
  int tc = 1; (Tests) && (cin >> tc);
  for(int i = 1; i <= tc; ++i)
    Solve();
}

int main()
{
  Fast(); initialize(); MultiTest(1);
}
```

## 4.2 Euler totient function

```cpp
int lp[N], Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

```

```cpp
ll Phi(ll a) { // for Queries
  ll ret = a, p;
  for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
    if (p * p > a) break;
    if (a % p) continue;
    ret -= ret / p;
    while (a % p == 0) a /= p;
  }
  if (a > 1) ret -= ret / a;
  return ret;
}
```

## 4.3 Extended wheel factorization

```cpp
/*
  Constraints:
  1 <= n <= 1e7
  2 <= a <= 1e{14}

  Time Complexity:
  linear_sieve takes O(n)
  Factorization takes O(n / (ln(n) - 1.08))

  Space Complexity:
  O(MaxN + n / (ln(n) - 1.08)
*/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

vector<pair<ll, int>> Factorization(ll a) {
  vector<pair<ll, int> > ret;
  ll p;
  for (int i = 0, cnt; i < pnx && (p = Primes[i], true) && p * p <= a; ++i) {
    if (a % p) continue;
    cnt = 0;
    while (a % p == 0) a /= p, ++cnt;
    ret.emplace_back(p, cnt);
  }
  if (a > 1) ret.emplace_back(a, 1);
  return ret;
}
```

## 4.4 Least prime factorization

```cpp
/*
  Constraints:
  1 <= n <= 1e7

  Time Complexity:
  linear_sieve takes O(n)
  Factorization takes O(log(n))

  Space Complexity:
  O(MaxN + n / (ln(n) - 1.08)
*/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

vector<pair<int, int>> Factorization(int n) {
  vector<pair<int, int>> ret;
  while (n > 1) {
    int p = leastPrime[n], cnt = 0;
    while (n % p == 0) n /= p, ++cnt;
    ret.emplace_back(p, cnt);
  }
  return ret;
}
```

## 4.5 Mobius function

```cpp
/**
    Constraints:
    1 <= x <= 1e7
    2 <= n <= 10^{14}

    Time Complexity:
    linear_sieve takes O(x)
    mobius takes O(n / (ln(n) - 1.08))

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))
*/

int lp[N], Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int x) {
  for (int i = 2; i <= x; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }

    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= x; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

int mobius(ll n) {
  ll p, pp;
  char mob = 1;
  for (int i = 0; i < pnx && (p = Primes[i], pp = p * p, true); ++i) {
    if (pp > n) break;
    if (n % p) continue;
    if (n % pp == 0) return 0;
    n /= p;
    mob = -mob;
  }
  if (n > 1) mob = -mob;
  return mob;
}
```

## 4.6 Phi factorial

```cpp
/**
    Constraints:
    1 <= x <= 1e7
    2 <= n <= 1e7

    Time Complexity:
    linear_sieve takes O(x)
    phi_factorial takes O(n)

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))
*/

int lp[N], Primes[664580], pnx; /** number of primes = n / (ln(n) - 1.08) **/

void linear_sieve(int x) {
  for (int i = 2; i <= x; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }

    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= x; ++j) {
      lp[comp] = Primes[j];
    }
  }
}
```

```
ll phi_factorial(int n) {
  ll ret = 1;
  for (int i = 2; i <= n; ++i) {
    ret = ret * (lp[i] == i ? i - 1 : i);
  }
  return ret;
}
```

## 4.7 Linear sieve

```
/**
    Constraints:
    1 <= n <= 1e7

    Time Complexity:
    linear_sieve takes O(n)

    Space Complexity:
    O(MaxN + n / (ln(n) - 1.08))
**/

int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}
```

## 4.8 Segmented sieve

```
int lp[N];
int Primes[664580], pnx; /** size of Primes = n / (ln(n) - 1.08) */
bool isPrime[N];

void linear_sieve(int n) {
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
    }
    for (int j = 0, comp; j < pnx && Primes[j] <= lp[i] && (comp = i * Primes[j]) <= n; ++j) {
      lp[comp] = Primes[j];
    }
  }
}

vector<ll> segmented_sieve(ll l, ll r) {
  l += l == 1;
  int limit = r - l + 1;
  vector<ll> ret;
  memset(isPrime, true, sizeof(isPrime));

  ll p;
  for (int i = 0; i < pnx && (p = Primes[i], true); ++i) {
    for (ll j = max(p * p, (l + p - 1) / p * p); j <= r; j += p)
      isPrime[j - l] = false;
  }

  for (int i = 0; i < limit; ++i)
    if (isPrime[i])
      ret.emplace_back(i + l);
  return ret;
}
```

## 4.9 Miller Rabin test

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

#include <bits/stdc++.h>

#define endl      '\n'

using namespace std;

typedef long long  ll;
typedef __int128 i128;

const int N = 1e6;

void Fast() {
  cin.sync_with_stdio(0);
  cin.tie(0);cout.tie(0);
}

ll ModExp(ll base, ll e, ll mod)
{
  ll result;
  base %= mod;

  for(result = 1; e; e >>= 1ll)
    {
      if(e & 1ll)
  result = ((i128)result * base) % mod;
      base = ((i128)base * base) % mod;
    }
  return result;
}

bool CheckComposite(ll n, ll p, ll d, int r)
{
  ll a = ModExp(p, d, n);
  if(a == 1 || a == n - 1)
    return false;

  for(int i = 1; i < r; ++i)
    {
      a = ((i128)a * a) % n;
      if(a == n - 1)
  return false;
    }
  return true;
}

bool Miller(ll n)
{
  if(n < 2) return false;

  int r; ll d;
  for(r = 0, d = n - 1; (d & 1ll) == 0; d >>= 1ll, ++r);

  for(int p : {2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 37})
    {
      if(n == p)
  return true;
      if(CheckComposite(n, p, d, r))
  return false;
    }
  return true;
}

int main()
{
  Fast();

  ll n;
  cin >> n;
  cout << (Miller(n) ? "Yes, it is Prime" : "No, it is not a prime") << endl;
}
```

## 4.10 Stable marriage problem

```
/** https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3616
 **/

#pragma GCC optimize ("Ofast")

#include <bits/stdc++.h>

#define endl      '\n'

using namespace std;

typedef int64_t    ll;
typedef __int128 i128;

void Fast() {
  cin.sync_with_stdio(0);
  cin.tie(0);cout.tie(0);
```

```
}

const int N = 1e3+ 9, M = 1e3 + 9, oo = 0x3f3f3f3f;
queue <int> Q;

int husband[N], wife[N], Next[N], order[N][N], pref[N][N], n, v;

void _clear() {
  memset(wife,    0, sizeof(wife[0])    * (n + 2));
  memset(husband, 0, sizeof(husband[0]) * (n + 2));
  memset(Next,    0, sizeof(Next[0])    * (n + 2));
}

void engage(int man, int woman) {
  int exWife = wife[man];
  wife[man] = woman;
  husband[woman] = man;

  if(exWife)
    Q.push(exWife);
}

void Solve()
{
  _clear();

  cin >> n;
  for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= n; ++j)
      cin >> pref[i][j];

  for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= n; ++j) {
      cin >> v;
      order[i][v] = j;
    }

  for(int i = 1; i <= n; ++i)
    Q.push(i);

  int man, woman;
  while(Q.size())
    {
      woman = Q.front(); Q.pop();
      man = pref[woman][++Next[woman]];

      if(!wife[man] || order[man][woman] < order[man][wife[man]])
  engage(man, woman);
      else
  Q.push(woman);
    }

  for(int i = 1; i <= n; ++i)
    cout << husband[i] << endl;
}

int main()
{
  Fast();

  int tc = 1; cin >> tc;
  for(int i = 1; i <= tc; ++i) {
    if(i > 1) cout << endl;
    Solve();
  }
}

/**
   1
   3
   3 2 1
   2 1 3
   2 1 3
   3 1 2
   1 2 3
   2 3 1
   -----
   3
   2
   1
**/
```

## 4.11 Euler totient sieve

```
/*
  Constraints:
  1 <= n <= 1e7

  Time Complexity:
  Phi_sieve takes O(n * ln(ln(n)))

  Space Complexity:
  MaxN
*/

int EulerPhi[N];

void Phi_sieve(int n) {
  for (int i = 1; i <= n; ++i) {
    EulerPhi[i] = i;
  }
  for (int i = 2; i <= n; ++i) {
    if (EulerPhi[i] == i)
      for (int j = i; j <= n; j += i) {
  EulerPhi[j] -= EulerPhi[j] / i;
      }
  }
}
```

## 4.12 Mobius sieve

```
/*
  Constraints:
  1 <= n <= 1e7

  Time Complexity:
  mu_sieve takes O(n)

  Space Complexity:
  O(MaxN)
*/

int mu[N], lp[N], Primes[78522], pnx;

void mu_sieve(int n) {
  mu[1] = 1;
  fill(mu, mu + N, 1);
  for (int i = 2; i <= n; ++i) {
    if (lp[i] == 0) {
      lp[i] = Primes[pnx++] = i;
      mu[i] = -1;
    }
    for (int j = 0, nxt; j < pnx && Primes[j] <= lp[i] && (nxt = i * Primes[j]) <= n; ++j) {
      lp[nxt] = Primes[j];
      mu[nxt] = (lp[i] == Primes[j] ? 0 : -mu[i]);
    }
  }
}
```

## 4.13 Wheel sieve

```
/**
    Constraints:
    1 <= n <= 1e9
    2 <= x <= 9700000

    Time Complexity:
    wheel_sieve takes O(n / ln(ln(n)))
    coPrimes takes O(x * ln(ln(x)))

    Space Complexity:
    O(MaxN / 32 + n / (ln(n) - 1.08) + x)
**/

bitset<N> isPrime;
int inx[30100];
int Primes[50908031], pnx; /** size of Primes = n / (ln(n) - 1.08) */

vector<int> coPrimes(int x) {
  int basis[5] = {3, 5, 7, 11, 13};

  vector<int> ret;
  bitset<30100> isCoprime;
  isCoprime.set();

  for (int b : basis)
    for (int d = b * b; d <= x; d += b << 1)
      isCoprime.reset(d);

```

```
  for (int i = 17; i <= x; i += 2)
    if (isCoprime[i]) ret.push_back(i);

  ret.push_back(x + 1);
  ret.push_back(x + 17);
  return ret;
}

void wheel_sieve(int n) {
  int basis[6] = {2, 3, 5, 7, 11, 13};
  vector<int> wheel = coPrimes(2 * 3 * 5 * 7 * 11 * 13);
  int sz = wheel.size();

  for (int k = 0; k < sz; ++k)
    inx[wheel[k]] = k;

  isPrime.set();
  inx[1] = sz - 2;
  int inc[sz - 1];

  for (int i = 1; i < sz; ++i)
    inc[i - 1] = wheel[i] - wheel[i - 1];

  for (int p : basis) {
    if (n >= p)
      Primes[pnx++] = p;
  }

  int c = 0;
  for (ll i = 17; i <= n; i += inc[c++]) {
    if (isPrime[i]) {
      Primes[pnx++] = i;
      int d = inx[i % 30030];
      for (ll j = i * i; j <= n; j += i * inc[d++]) {
  isPrime.reset(j);
  if (d == sz - 1) d = 0;
      }
    }
    if (c == sz - 1) c = 0;
  }
}
```

# 5 String Processing

## 5.1 Trie

```
class Trie {
private:
  Trie* children[26]; // Pointer = 8 Byte; 8*26 = 208 Byte
  int prefixs, words; // 8 Byte
  bool iseow; // 1 Byte
  char cur_letter; // 1 Byte
  vector <string> lex;
  priority_queue <pair <int, string>, vector <pair <int, string>>, greater <pair <int, string>>>
        occurrence; // small at top

public:
  Trie(char lett = '\0') {
    memset(children, 0, sizeof(children));
    prefixs = words = 0;
    iseow = false;
    cur_letter = lett;
  }

  void insert(string &str) { // O(l)
    Trie* cur = this;
    int inx, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(cur->children[inx] == nullptr)
  cur->children[inx] = new Trie(str[i]);

      cur = cur->children[inx];
      cur->prefixs++;
    }
    cur->iseow = true;
    cur->words++;
  }

  int search_word(string &str) { // O(l)
    Trie* cur = this;
    int inx, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(cur->children[inx] == nullptr) {
  return 0;
      }
      cur = cur->children[inx];
    }
    return cur->words;
  }

  int search_prefix(string &str) { // O(l)
    Trie* cur = this;
    int inx = 0, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(cur->children[inx] == nullptr) {
  return 0;
      }
      cur = cur->children[inx];
    }
    return cur->prefixs;
  }

  bool erase(string &str) {
    if(!search_word(str))
      return false;

    Trie* cur = this;
    int inx, strsz = str.size();
    for(int i = 0; i < strsz; ++i) {
      inx = str[i] - 'a';
      if(--cur->children[inx]->prefixs == 0) {
  cur->children[inx] = nullptr;
  return true;
      }
      cur = cur->children[inx];
    }
    if(--cur->words == 0) {
      cur->iseow = false;
    }
    return true;
  }

private:
  void dfs(Trie* node, string s) { // lex order dfs -> traverse all the strings starting from root
    if(node->iseow) {
      lex.emplace_back(s);
    }

    for(int j = 0; j < 26; ++j)
      if(node->children[j] != nullptr) {
  dfs(node->children[j], s + string(1, node->children[j]->cur_letter));
      }
  }

  void dfs2(Trie* node, string s) { // autocomplete dfs -> traverse all the strings starting from the
        end of the given prefix
    if(node->iseow) {
      if(occurrence.size() < 10) {
  occurrence.push(make_pair(node->words, s));
      } else {
  if(node->words > occurrence.top().first) {
    occurrence.pop();
    occurrence.push(make_pair(node->words, s));
  }
      }
    }

    for(int i = 0; i < 26; ++i) if(node->children[i] != nullptr) {
  dfs2(node->children[i], s + string(1, node->children[i]->cur_letter));
      }
  }

public:
  vector <string> lex_order() { // all strings in lexicographical order
    lex.clear();
    Trie* cur = this;
    for(int i = 0; i < 26; ++i) if(cur->children[i] != nullptr) {
  dfs(cur->children[i], string(1, cur->children[i]->cur_letter));
      }
    return lex;
  }

  void autocomplete(string &pref) { // suggest top ten words with max frequency
    if(!search_prefix(pref))
      return;

    Trie* cur = this;
    int inx, presz = pref.size();
    for(int i = 0; i < presz; ++i) {
      inx = pref[i] - 'a';
      cur = cur->children[inx];
    }
```

```cpp
    for(int i = 0; i < 26; ++i) if(cur->children[i] != nullptr) {
  dfs2(cur->children[i], string(1, cur->children[i]->cur_letter));
      }

    vector <string> st;
    while(!occurrence.empty()) {
      st.emplace_back(pref + occurrence.top().second);
      occurrence.pop();
    }
    if(cur->iseow) {
      st.emplace_back(pref);
    }
    while(!st.empty()) {
      cout << st.back() << endl;
      st.pop_back();
    }
  }
};
```

## 5.2 Knuth Morris Pratt (KMP)

```cpp
/**
 * KMP(Knuth-Morris-Pratt) Algorithm
 ** Longest Prefix
 *** proper prefix = all prefixes except the whole string
 *** propre suffix = all suffixes except the whole string
 ** Prefix Function = Failure Function
 *** Given String P of len m, Find F[m];
 *** let t = P[0....i]
 *** f[i] = length of the longest proper prefix of t that is suffix of t
 *** calculating i different ways
 *** match the pattern against itself
 *** O(m) for failure function
 *** O(n) for KMP
 **/

vector <int> LongestPrefix(string &p) {
  int psz = p.size();
  vector <int> longest_prefix(psz, 0);

  for(int i = 1, k = 0; i < psz; ++i) {
    while(k && p[k] != p[i]) k = longest_prefix[k - 1];
    longest_prefix[i] = (p[k] == p[i] ? ++k : k);
  }
  return longest_prefix;
}

vector <int> KMP(string &s, string &p) {
  int ssz = s.size(), psz = p.size();

  vector <int> longest_prefix = LongestPrefix(p), matches;

  for(int i = 0, k = 0; i < ssz; ++i) {
    while(k && p[k] != s[i]) k = longest_prefix[k - 1]; // Fail go back
    k += (p[k] == s[i]);

    if(k == psz) {
      matches.emplace_back(i - psz + 1);
      k = longest_prefix[k - 1]; // faill safe and find another pattern
    }
  }
  return matches;
}
```

# 6 Geometry

## 6.1 Point

```cpp
class point
{
public :
  ld x, y;

  point() = default;
  point(ld _x, ld _y) : x(_x), y(_y) {}

  bool operator < (point other) const {
    if(fabs(x - other.x) > EPS) // if(x != other.x)
      return x < other.x;
    return y < other.y;
  }

  bool operator == (point other) const {
    return ((fabs(x - other.x) < EPS) && (fabs(y - other.y) < EPS)); // " < EPS " equal to " == zero "
  }

  bool operator > (point other) const {
    if(fabs(x - other.x) > EPS)
      return x > other.x;
    return y > other.y;
  }

  ld dist(point other) { // Euclidean distance
    ld dx = this->x - other.x;
    ld dy = this->y - other.y;
    return sqrtl(dx * dx + dy * dy);
  }

  ld DEG_to_RAD(ld theta) {
    return theta * PI / 180.0;
  }

  ld RAD_to_DEG(ld theta) {
    return theta * 180.0 / PI;
  }

  point rotate(ld theta) {
    ld rad = DEG_to_RAD(theta);
    return point(cos(theta) * x - sin(theta) * y,
     sin(theta) * x + cos(theta) * y);
  }
};
```

# 7 More advanced topics

## 7.1 A*-Algorithm

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

#include <bits/stdc++.h>

#define endl    '\n'

using namespace std;

typedef int64_t  ll;

void Fast() {
    cin.sync_with_stdio(0);
    cin.tie(0);cout.tie(0);
}

const int dr []     = {-1, 0, 1, 0};
const int dc []     = {0, 1, 0, -1};
const char dir []   = {'U', 'R', 'D', 'L'};
map <char, int> inv = { {'U', 0}, {'R', 1}, {'D', 2}, {'L', 3}};

const int N = 1e3 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f;
const ll INF = 0x3f3f3f3f3f3f3f3f;

char grid[N][N];
int dis[N][N], n, m, si, sj, ti, tj;
char Par[N][N];

vector < pair <int, int> > restorePath(int sr, int sc, int tr, int tc)
{
    vector < pair <int, int> > ret;
    if(dis[tr][tc] == oo) return ret;

    for(char i = Par[tr][tc]; (sr ^ tr) || (sc ^ tc); i = Par[tr][tc])
    {
        ret.push_back({tr, tc});
        tr += dr[inv[i]];
        tc += dc[inv[i]];
    }

    ret.push_back({sr, sc});
    reverse(ret.begin(), ret.end());
    return ret;
}

bool valid(int r, int c) {
    return r >= 0 && r < n && c >= 0 && c < m && grid[r][c] != '%';
```

```cpp
}

/** admissible heuristic **/
int manhattanDistance(int x1, int y1, int x2, int y2) {
    return (abs(x1 - x2) + abs(y1 - y2));
}

int Astar(int sr, int sc, int tr, int tc)
{
    memset(dis, 0x3f, sizeof (dis));
    memset(Par,   -1, sizeof (Par));

    priority_queue <tuple <int, int, int> > Q;

    dis[sr][sc] = 0;
    Q.push({-manhattanDistance(sr, sc, tr, tc), sr, sc});

    int hcost, r, c, nr, nc;
    while(Q.size())
    {
        tie(hcost, r, c) = Q.top(); Q.pop();
        if(r == tr && c == tc) return dis[r][c];

        for(int i = 0; i < 4; ++i)
        {
            nr = r + dr[i];
            nc = c + dc[i];

            if(!valid(nr, nc)) continue;

            if(dis[r][c] + 1 < dis[nr][nc])
            {
                dis[nr][nc] = dis[r][c] + 1;
                Par[nr][nc] = dir[i ^ 2];
                Q.push({-dis[nr][nc] -manhattanDistance(nr, nc, tr, tc), nr, nc});
            }
        }
    }
    return -1;
}

void Solve()
{
    Fast();

    cin >> si >> sj >> ti >> tj >> n >> m;

    for(int i = 0; i < n; ++i)
        for(int j = 0; j < m; ++j)
            cin >> grid[i][j];

    cout << Astar(si, sj, ti, tj) << endl;
    vector < pair <int, int> > path = restorePath(si, sj, ti, tj);

    for(auto point : path)
        cout << point.first << " " << point.second << endl;
}

int main()
{
    int t = 1;
    while(t--) Solve();
}

/**
    P -> strat
    . -> target

    input:
    0 2 2 3 5 5
    %%P%-
    -%---
    %--.-
    %%%%%
    -----

    output:
    3
    0 2
    1 2
    2 2
    2 3

**/
```

## 7.2 Mo's algorithm

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

#include <bits/stdc++.h>

#define endl        '\n'

using namespace std;

typedef int64_t    ll;
typedef __int128 i128;

void Fast() {
  cin.sync_with_stdio(0);
  cin.tie(0);cout.tie(0);
}

void File() {
  freopen("input.in",  "r", stdin);
  freopen("output.out", "w", stdout);
}

const int N = 3e4 + 9, M = 2e5 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ll INF = 0x3f3f3f3f3f3f3f3f;
const int BLK = 256;

struct query
{
  int l, r, id, blk;

  query() = default;
  query(int _l, int _r, int _id) {
    l = _l;
    r = _r;
    id = _id;
    blk = l / BLK;
  }

  bool operator < (const query other) const {
    if(blk ^ other.blk)
      return blk < other.blk;
    return (blk & 1) ? r < other.r : r > other.r;
  }
} queries[M];

int res[M], freq[M << 3], cur;

void add(int id) {
  cur += (++freq[id] == 1);
}

void remove(int id) {
  cur -= (--freq[id] == 0);
}

int get_res() {
  return cur;
}

int cur_l, cur_r, l, r, n, q, a[N];

void Solve()
{
  cin >> n;
  for(int i = 1; i <= n; ++i) cin >> a[i];

  cin >> q;
  for(int i = 1; i <= q; ++i) {
    cin >> l >> r;
    queries[i] = query(l, r, i);
  }

  sort(queries + 1, queries + 1 + q);

  cur_l = 1, cur_r = 0; // assign to right invalid index
  for(int i = 1; i <= q; ++i)
    {
      int ql = queries[i].l;
      int qr = queries[i].r;

      // Add right
      while(cur_r < qr) add(a[++cur_r]);
      // Add left
      while(cur_l > ql) add(a[--cur_l]);
      // Remove right
      while(cur_r > qr) remove(a[cur_r--]);
      // Remove left
      while(cur_l < ql) remove(a[cur_l++]);

      res[queries[i].id] = get_res();
    }

  for(int i = 1; i <= q; ++i)
```

```
    cout << res[i] << "\n";
}

int main()
{
  Fast();

  int tc = 1;
  for(int i = 1; i <= tc; ++i)
    Solve();
}
```

## 7.3 Square root decomposition

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

#include <bits/stdc++.h>

#define endl        '\n'

using namespace std;

typedef int64_t    ll;
typedef __int128 i128;

void Fast() {
  cin.sync_with_stdio(0);
  cin.tie(0);cout.tie(0);
}

const int N = 5e5 + 9, M = 1e3 + 9, oo = 0x3f3f3f3f, Mod = 1e9 + 7;
const ll INF = 0x3f3f3f3f3f3f3f3f;
const int BLK = 256;

int n, q, a[N], type, x, y, z;
vector <int> bs[M];

int query(int l, int r, int val)
{
  int cur_l = l / BLK;
  int cur_r = r / BLK;
  int ans = 0;

  if(cur_l == cur_r) {
    for (int i = l; i <= r; ++i)
      ans += (a[i] >= val);
  } else {
    for(int i = l, _end = (cur_l + 1) * BLK; i < _end; ++i)
      ans += (a[i] >= val);
    for(int i = cur_l + 1; i <= cur_r - 1; ++i)
      ans += bs[i].end() - lower_bound(bs[i].begin(), bs[i].end(), val);
    for(int i = cur_r * BLK; i <= r; ++i)
      ans += (a[i] >= val);
  }
  return ans;
}

void build()
{
  for(int i = 0; i < n; ++i)
    bs[i / BLK].emplace_back(a[i]);

  for(int i = 0; i < M; ++i)
    sort(bs[i].begin(), bs[i].end());
}

void update(int id, int delta)
{
  int pos = lower_bound(bs[id / BLK].begin(), bs[id / BLK].end(), a[id]) - bs[id / BLK].begin();
  bs[id / BLK][pos] = delta;
  sort(bs[id / BLK].begin(), bs[id / BLK].end());
  a[id] = delta;
}

void Solve()
{
  cin >> n;
  for(int i = 1; i <= n; ++i) cin >> a[i];

  build();

  cin >> q;
  while(q--)
    {
      cin >> type >> x >> y;
      if(type == 0)
        {
    cin >> z;
    cout << query(x, y, z) << endl;
        }
      else
  update(x, y);
    }
}

int main()
{
  Fast();

  int tc = 1;
  for(int i = 1; i <= tc; ++i)
    Solve();
}
```

# 8 Miscellaneous

## 8.1 Double comparison

```
bool approximatelyEqual(double a, double b, double epsilon)
{
  return fabs(a - b) <= ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool essentiallyEqual(double a, double b, double epsilon)
{
  return fabs(a - b) <= ((fabs(a) > fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool definitelyGreaterThan(double a, double b, double epsilon)
{
  return (a - b) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}

bool definitelyLessThan(double a, double b, double epsilon)
{
  return (b - a) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

## 8.2 Fast input output

```
/**
   Fast Input/Output method for C++:
   1. cin(with sync_with_stdio(false) & cin.tie(nullptr)):
   - int:
   - |n = 5e6| => 420ms
   - |n = 1e7| => 742ms
   - ll:
   - |n = 5e6| => 895ms

   2. read (using getchar()):
   - int:
   - |n = 5e6| => 173ms
   - |n = 1e7| => 172ms
   - ll:
   - |n = 5e6| => 340ms
**/

ll readll () {
  bool minus = false;
  unsigned long long result = 0;
  char ch;
  ch = getchar();

  while (true) {
    if (ch == '-') break;
    if (ch >= '0' && ch <= '9') break;
    ch = getchar();
  }

  if (ch == '-') minus = true;
  else result = ch - '0';

  while (true) {
    ch = getchar();
    if (ch < '0' || ch > '9') break;
    result = result * 10 + (ch - '0');
```

```
  }

  if (minus) return -(ll)result;
  return result;
}

int readi () {
  bool minus = false;
  unsigned int result = 0;
  char ch;
  ch = getchar();

  while (true) {
    if (ch == '-') break;
    if (ch >= '0' && ch <= '9') break;
    ch = getchar();
  }

  if (ch == '-') minus = true;
  else result = ch - '0';

  while (true) {
    ch = getchar();
    if (ch < '0' || ch > '9') break;
    result = result * 10 + (ch - '0');
  }

  if (minus) return -(int)result;
  return result;
}
```

## 8.3 Gcd & Lcm

```
ll gcd(ll a, ll b) { // binary GCD uses about 60% fewer bit operations
  if (!a) return b;

  int shift = __builtin_ctz(a | b);
  a >>= __builtin_ctz(a);

  while (b) {
    b >>= __builtin_ctz(b);

    if (a > b)
      swap(a, b);
    b -= a;
  }
  return a << shift;
}

ll lcm(ll a, ll b) {
  return a / gcd(a, b) * b;
}
```

## 8.4 Modular calculations

```
/*
  - 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:
*/

// 1. Modular Exponentiation

ll binExp(ll a, ll b, ll p) {
  ll res = 1;
  while (b) {
    if (b & 1ll)
      res = res * a % p;
    a = a * a % p;
    b >>= 1;
  }
  return res;
}

// 2. Modular Multiplication

ll binMul(ll a, ll b, ll p) {
  ll res = 0;
  a %= p;
  while (b) {
    if (b & 1ll)
      res = (res + a) % p;
    a = (a + a) % p;
    b >>= 1;
  }
  return res;
}

// 3. Modular Multiplicative Inverse

ll modInv(ll b, ll p) {
  return binExp(b, p - 2, p); // Guaranteed that p is a Prime Number
}
```

## 8.5 Overloaded operators to accept 128 Bit integer

```
typedef __uint128_t       ui128;
typedef __int128           i128;

template <class T> string to_string(T x)
{
  int sn = 1; if(x < 0) sn = -1, x *= sn; string s = "";
  do { s = "0123456789"[x % 10] + s, x /= 10; } while(x);
  return (sn == -1 ? "-" : "") + s;
}

auto str_to_int(string x)
{
  ui128 ret = (x[0] == '-' ? 0 : x[0] - '0');
  for(int i = 1; i < x.size(); ++i) ret = ret * 10 + (x[i] - '0');
  return (x[0] == '-' ? -1 * (i128)ret : ret);
}

istream & operator >> (istream & in, i128 & i) noexcept { string s; in >> s; i = str_to_int(s); return in; }
ostream & operator << (ostream & os, const i128 i) noexcept { os << to_string(i); return os; }
istream & operator >> (istream & in, ui128 & i) noexcept { string s; in >> s; i = str_to_int(s); return in; }
ostream & operator << (ostream & os, const ui128 i) noexcept { os << to_string(i); return os; }
```

## 8.6 Policy based data structures

```
#if      __cplusplus >= 201402L
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#endif

#if   __cplusplus >= 201402L
using namespace __gnu_cxx;
using namespace __gnu_pbds;
#endif

template <class T, typename Comp = less <T> >
using indexed_set = tree <T, null_type, Comp, rb_tree_tag, tree_order_statistics_node_update>;

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

## 8.7 stress test

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

/** 64-bit signed int Generator
 **/
i64 int64(i64 a, i64 b) {
  return uniform_int_distribution <i64> (a, b)(rng);
}

/** Customize your Generator depending on the input
 **/
void gen () {
  ofstream cout("input.in");
  i32 t = 2;
  cout << t << endl;

  while (t--) {
    i32 n = int64(1, 100), m = int64(1, 100);
    cout << n << " " << m << endl;

    while (m--) {
      i32 u = int64(1, n), v = int64(1, n), c = int64(1, 4);
      cout << u << " " << v << " " << c << endl;
```

```cpp
    }
  }
}

i32 main (i32 arg, char* args[]) {
  fast();

  i32 tc = 0;
  i32 limit = 100;
  if(arg != 3) return 0;

  string flags = "g++ -Wall -Wextra -Wshadow -Og -g -Ofast -std=c++17 -D_GLIBCXX_ASSERTIONS -DDEBUG -ggdb3 -fsanitize=address,undefined -fmax-errors=2 -o ";
  string ex = ".cpp", bf, oz, pr;

  bf = flags + args[1] + " " + args[1] + ex;
  oz = flags + args[2] + " " + args[2] + ex;
  char bff[bf.size() + 1];
  char ozz[oz.size() + 1];
  strcpy(bff, bf.c_str());
  strcpy(ozz, oz.c_str());

  // compile command
  system(bff);
  system(ozz);

  ex = ".out";
  pr = "./";
  bf = pr + args[1] + " < input.in > " + args[1] + ex;
  oz = pr + args[2] + " < input.in > " + args[2] + ex;
  strcpy(bff, bf.c_str());
  strcpy(ozz, oz.c_str());

  while (++tc <= limit) {
    gen();
    cerr << tc << endl;
    // run command
    system(bff);
    system(ozz);

    ifstream brute_forces("brute_force.out");
    ifstream optimizes("optimized.out");

    string brute_force, optimized;
    getline(brute_forces, brute_force, (char)EOF);
    getline(optimizes, optimized, (char)EOF);

    if(brute_force != optimized) {
      cerr << "Wrong Answer" << endl;
      break;
    } else if (tc == limit) {
      cout << "Accepted" << endl;
    }
  }
}
```