Contents

➢ Template

```
#include "bits/stdc++.h"

using namespace std;
typedef long long ll;
#define endl '\n'
#define int ll
#define Hi ios_base::sync_with_stdio(0);
cin.tie(0); cout.tie(0);
////////////////////////////////////////////////////////////
///

void magic() {}

signed main() {
  Hi
  int t = 1;
  cin >> t;
  while (t--) magic();
}
```

➢ Pragma

```
#pragma GCC optimize("O3")
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
#pragma GCC target("avx,avx2,fma")
```

➢ Custom Hash

```
struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().co
unt();
    return splitmix64(x + FIXED_RANDOM);
  }
};
```

➢ Cycle Take All Edges

```
vector<int> res, vis;
vector<vector<pair<int, int>>> adj;

void dfs(int u) {
  while (adj[u].size()) {
    auto [v, i] = adj[u].back();
    adj[u].pop_back();
    if (vis[i])
      continue;
    vis[i] = 1;
    dfs(v);
  }
  res.push_back(u);
}
```

➢ Path Take All Edges

```
vector<int> res;
vector<vector<int>> adj;
void dfs(int s){
  while(adj[s].size()){
    int u = adj[s].back(); adj[s].pop_back();
    dfs(u);
  }
  res.push_back(s);
}

void check() {

  for(int i = 2; i < n; i++)
    if(in[i] != out[i])
      return "IMPOSSIBLE";

  if(in[1] != out[1] - 1 || out[n] != in[n] - 1)
    return "IMPOSSIBLE";

  dfs(1);
  reverse(res.begin(), res.end());
  if(res.size() != m + 1 || res.back() != n) {
    return "IMPOSSIBLE";

  for(auto v : res)
    cout << v << " ";
}
```

➢ De Burijn

```
unordered_map<string, int> id;
string s[MX_SIZE];
int cnt = 1, n, k = 2;
vector<char> a{'0', '1'};

void rec(string cur){
  if(cur.size() == n){
    id[cur] = cnt;
    s[cnt++] = cur;
    return;
  }

  for(int i = 0; i < a.size(); i++)
    rec(cur + a[i]);
}

vector<int> adj[MX_SIZE];
bool vis[MX_SIZE];
string res;
void dfs(int u) {
  for (auto v: adj[u])
    if (!vis[v]) {
      vis[v] = 1;
      dfs(v);
      res.push_back(s[v].back());
    }
}

void makeAdj(){
  int mx = (1<<n);
  for(int i = 1; i <= mx; i++){
    string cur = s[i].substr(1, n - 1);
    for(int j = 0; j < a.size(); j++)
      adj[i].push_back(id[cur + a[j]]);
  }
}

string makeDeBruijn() {
  rec("");
  makeAdj();

  dfs(1);
  s[1].pop_back();
  return res + s[1];
}
```

➢ Find Cycle

```
int anyCycle = 0;
vector<int> cycle;

void dfs(int u) {
  vis[u] = tc;
  for (auto v: adj[u]) {
    if (anyCycle == tc)
      return;

    if (vis[v] != tc) {
      parent[v] = u;
      dfs(v);
    } else if (v != parent[u]) {
      cycle.push_back(v);

      int current = u;
      while (current != v) {
        cycle.push_back(current);
        current = parent[current];
      }
      cycle.push_back(v);

      anyCycle = tc;
      return;
    }
  }
}
```

➢ Find Cycle Directed

```
int anyCycle = 0;
vector<int> cycle;
int n, m, in = 1, out = 2;

void dfs(int u) {
  vis[u] = in;
  for (auto v: adj[u]) {
    if (anyCycle == 1)
      return;

    if (vis[v] != in && vis[v] != out ) {
      parent[v] = u;
      dfs(v);
    } else if (vis[v] != out) {
      cycle.push_back(v);

      int current = u;
      while (current != v) {
        cycle.push_back(current);
        current = parent[current];
      }
      cycle.push_back(v);

      anyCycle = 1;
      return;
    }
  }

  vis[u] = out;
}
```

➢ Tree Canonical Form

```
string treeCanonicalForm(int u, int par) {
  vector<string> child;

  for(auto v : adj[u]){
    if(v == par) continue;
    child.push_back(treeCanonicalForm(v, u));
  }

  string cur = "(";
  sort(child.begin(), child.end());
  for(auto it : child)
    cur += it;
  cur += ")";

  return cur;
}
```

➢ Tree Canonical BFS

```
string calc(int u, vector<vector<string>> &subCan) {
  string cur = "(";
  sort(subCan[u].begin(), subCan[u].end());
  for (auto it: subCan[u])
    cur += it;
  cur += ")";

  return cur;
}

pair<string, string> treeCanonicalForm(int n) {
  queue<int> leaf;
  vector<int> deg(n + 1, -1);
  int rem = n;

  for (int i = 1; i <= n; i++) {
    if (adj[i].size() <= 1)
      leaf.push(i);
    else
      deg[i] = adj[i].size();
  }

  vector<vector<string>> subCan(n + 1);
  while (rem > 2) {
    int v = leaf.front();
    leaf.pop();
    string cur = calc(v, subCan);
    for (auto u: adj[v]) {
      subCan[u].push_back(cur);
      if (--deg[u] == 1)
        leaf.push(u);
    }
    rem--;
  }

  int v1 = leaf.front();
  leaf.pop();
  int v2 = (leaf.empty() ? -1 : leaf.front());

  string s1 = calc(v1, subCan);
  string s2 = ((v2 == -1) ? "" : calc(v2, subCan));

  if (v2 == -1)
    return make_pair(s1, s2);

  subCan[v1].push_back(s2);
  subCan[v2].push_back(s1);
  return {calc(v1, subCan), calc(v2, subCan)};
}
```

➢ Tree Diameter

```
void dfsDown(int u, int p) {
  for (auto v: adj[u]) {
    if (p == v)
      continue;

    dfsDown(v, u);
    if (dpDown[v][0] + 1 > dpDown[u][1])
      dpDown[u][1] = dpDown[v][0] + 1;
    if (dpDown[u][1] > dpDown[u][0])
      swap(dpDown[u][0], dpDown[u][1]);
  }

}

void dfsUp(int u, int p, int level) {
  dpUp[u] = level;
  for (auto v: adj[u]) {
    if (p == v)
      continue;

    int cur;
    if (dpDown[v][0] + 1 != dpDown[u][0])
      cur = dpDown[u][0];
    else
      cur = dpDown[u][1];
    cur = max(cur, dpUp[u]);

    dfsUp(v, u, cur + 1);
  }
}
```

➢ DSU

```
struct DSU {
  int n, cnt;
  vector<int> parent, size;

  DSU(int sz) {
    n = sz, cnt = 0;
    parent.resize(n);
    size.resize(n);
  }

  void init() {
    for(int u = 0; u < n; ++u) makeSet(u);
  }

  void makeSet(int u) {
    if(!size[u]) cnt++;
    parent[u] = u;
    size[u] = 1;
  }

  int find(int u) {
    if(u == parent[u]) return u;
    return parent[u] = find(parent[u]);
  }

  bool merge(int u, int v) {
    int uP = find(u);
    int vP = find(v);

    if(vP == uP) return 0;

    if(size[vP] > size[uP]) swap(uP, vP);
    size[uP] += size[vP];
    parent[vP] = uP;
    --cnt;
    return 1;
  }
};
```

➢ Bellman

```
struct Edge {
  int from, to, w;
  Edge() : to(0), from(0), w(0) {};
  Edge(int f, int t, int ww) : from(f), to(t), w(ww) {};
};

struct Node {
  long long dist;
  int par;
  bool reachCycle;
  Node() : dist(oo), par(-1), reachCycle(0) {};
};

struct bellmanFord {
  int n, m;
  bool negative_cycle = 0;
  vector<Edge> edges;
  vector<Node> node;

  bellmanFord(int nn, int mm, vector<Edge> &e) {
    n = nn, m = mm;
    edges.resize(m);
    node.resize(n + 2);
    edges = e;
  }

  void build(int src, int startCost) {
    node[src].dist = startCost;
    node[src].par = src;

    for (int i = 1; i <= n; ++i) {
      bool exist = 0;
      for (long long j = 0; j < m; ++j) {
        auto [u, v, w] = edges[j];

        if (node[u].dist < oo && node[v].dist >
node[u].dist + w) {
          if (i == n) negative_cycle = 1;
          node[v].dist = node[u].dist + w;
          node[v].par = u;
          exist = 1;
        }
      }
      if (!exist) break;
    }
  }

  void buildReachCycle(int src) {
    vector<long long> oldDist(n + 1);
    for (int i = 1; i <= n; i++)
      oldDist[i] = node[i].dist;

    build(src, oldDist[src]);
    for (int i = 1; i <= n; i++)
      node[i].reachCycle = (oldDist[i] != node[i].dist);
  }
};
```

➢ Floyd

```
struct floyd {
  int n;
  vector<vector<int>> dist, par;

  floyd(int sz) {
    n = sz;
    dist = par = vector<vector<int>>(n + 5, vector<int>(n + 5));
    init();
  }

  void init() {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++)
        par[i][j] = i, dist[i][j] = oo;
      dist[i][i] = 0;
    }
  }

  void build() {
    for (int k = 1; k <= n; k++)
      for (int u = 1; u <= n; u++)
        for (int v = 1; v <= n; v++) {
          dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v]);
          if (dist[u][v] == dist[u][k] + dist[k][v] && k != v)
            par[u][v] = par[k][v];
        }
  }

  void printPath(int u, int v) {
    if (u != v) printPath(u, par[u][v]);
    cout << v << " ";
  }

  bool isNegativeCycle() {
    for (int i = 1; i <= n; i++)
      if (dist[i][i] < 0)
        return 1;
    return 0;
  }

  bool anyEffectiveNegativeCycle(int u, int v) {
    for (int i = 1; i <= n; i++)
      if (dist[i][i] < 0 && dist[u][i] < oo && dist[i][v] < oo)
        return 1;
    return 0;
  }
};
```

➢ Maximum Matching

```
struct maximumMatching {
  int n, t;
  vector<vector<int>> adj;
  vector<pair<int, int>> matches;
  vector<int> vis, leftAssign, rightAssign, isLeft,
isRight;

  maximumMatching(int nn) {
    n = nn, t = 0;
    adj.assign(n + 1, {});
    vis = isLeft = isRight = vector<int>(n + 1, 0);
    leftAssign = rightAssign = vector<int>(n + 1, -1);
  }

  void matching() { // O(EV)
    if (n == 0) return;

    int maxFlow = 0;
    for (int i = 1; i <= n; i++) {
      if (!isLeft[i]) continue;
      t++;
      if (canMatch(i)) maxFlow++;
    }

    for (int i = 1; i <= n; i++) {
      if (!isRight[i]) continue;


      if (rightAssign[i] != -1)
        matches.push_back({rightAssign[i], i});
    }
  }

  bool canMatch(int u) {
    for (auto v: adj[u])
      if (vis[v] != t) {
        vis[v] = t;
        if (rightAssign[v] == -1 ||
canMatch(rightAssign[v])) {
          rightAssign[v] = u, leftAssign[u] = v;
          return true;
        }
      }
    return false;
  }

  vector<vector<int>> mnPathCvs;
  void buildMnPathCvs() {
    for (int i = 1; i <= n; i++) {
      if (rightAssign[i] == -1) {
        vector<int> v(1, i);
        int cur = leftAssign[i];

        while (cur != -1) {
          v.push_back(cur);
          cur = leftAssign[cur];
        }
        mnPathCvs.push_back(v);
      }
    }
  }
};
```

➢ Max Flow (Dinic)

```
// O(v ^ 2 * E)
struct edge {
  int u, v, cap, flow;
};

struct Dinic {
  int n;
  vector<int> dist, par, idEdge, curId;
  vector<vector<int>> adj;
  vector<edge> edges;

  Dinic(int nn) {
    n = nn;
    dist.resize(n + 1);
    adj.resize(n + 1);
    par.resize(n + 1);
    idEdge.resize(n + 1);
    curId.resize(n + 1);
  }

  void addEdge(int u, int v, int c) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v, c, 0});
    adj[v].push_back(edges.size());
    edges.push_back({v, u, 0, 0});
  }

  int flow(int src, int sink) {
    int totalFlow = 0;
    while (isPath(src, sink)) {
      for (int i = 1; i <= n; i++)
        curId[i] = 0;
      while (int newFlow = dfs(src, oo, sink))
        totalFlow += newFlow;
    }

    return totalFlow;
  }

  bool isPath(int src, int sink) {
    for (int i = 1; i <= n; i++)
      dist[i] = oo;

    queue<int> q;
    q.push(src);
    dist[src] = 0;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      if (u == sink)
        break;

      for (auto idx: adj[u]) {
        auto [_, v, c, f] = edges[idx];
        if (f < c && dist[v] == oo) {
          dist[v] = dist[u] + 1;
          par[v] = u;
          idEdge[v] = idx;
          q.push(v);
        }
      }
    }

    return dist[sink] < oo;
  }

  int dfs(int u, int flow, int sink) {
    if (!flow)
      return 0;
    if (u == sink)
      return flow;
    for (; curId[u] < adj[u].size(); curId[u]++) {
      int idx = adj[u][curId[u]];
      auto [_, v, c, f] = edges[idx];

      if (dist[v] != dist[u] + 1)
        continue;
      int newFlow = dfs(v, min(flow, c - f), sink);
      if (newFlow) {
        edges[idx].flow += newFlow;
        edges[idx ^ 1].flow -= newFlow;
        return newFlow;
      }
    }
    return 0;
  }
};
```

➢ Max Flow (Karp)

```
struct edge {
  int u, v, cap;
};

struct maxFlow {
  int n;
  vector<int> dist, par, idEdge, vis;
  vector<vector<int>> adj;
  vector<edge> edges;

  maxFlow(int nn) {
    n = nn;
    dist.resize(n + 1);
    adj.resize(n + 1);
    par.resize(n + 1);
    vis.resize(n + 1, 0);
    idEdge.resize(n + 1);
  }

  void addEdge(int u, int v, int c) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v, c});
    adj[v].push_back(edges.size());
    edges.push_back({v, u, 0});
  }

  int flow(int src, int sink) {
    int totalFlow = 0;

    while (isPath(src, sink)) {
      int newFlow = oo;
      int cur = sink;
      while (cur != src) {
        newFlow = min(newFlow,
edges[idEdge[cur]].cap);
        cur = par[cur];
      }

      cur = sink;
      while (cur != src) {
        edges[idEdge[cur]].cap -= newFlow;
        edges[idEdge[cur] ^ 1].cap += newFlow;
        cur = par[cur];
      }

      totalFlow += newFlow;
    }
    return totalFlow;
  }

  bool isPath(int src, int sink) {
    for (int i = 1; i <= n; i++)
      dist[i] = oo;

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

    while (!q.empty()) {
      int u = q.front();
      q.pop();

      if (u == sink)
        break;

      for (auto idx: adj[u]) {
        auto [_, v, c] = edges[idx];
        if (c && dist[v] == oo) {
          dist[v] = dist[u] + 1;
          par[v] = u;
          idEdge[v] = idx;
          q.push(v);
        }
      }
    }

    return dist[sink] < oo;
  }

  vector<pair<int, int>> res;
  void minCutEdges(int src, int sink) {
    dfs(src);
    for (int idx = 0; idx < edges.size(); idx += 2) {
      auto [u, v, c] = edges[idx];
      if (vis[u] && !vis[v]) {
        res.push_back({u, v});
      }
    }
  }
```

```
  void dfs(int u) {
    vis[u] = 1;
    for (auto idx: adj[u]) {
      auto [_, v, c] = edges[idx];
      if (c && !vis[v]) {
        dfs(v);
      }
    }
  }
};
```

➢ Min Cost Max Flow

```
struct edge {
  int u, v, cost, cap;
};

struct minCostMaxFlow {
  int n, maxFlow, minCost;
  vector<int> dist, par, idEdge;
  vector<vector<int>> adj;
  vector<edge> edges;

  minCostMaxFlow(int nn) {
    n = nn;
    dist.resize(n + 1, oo);
    adj.resize(n + 1);
    par.resize(n + 1);
    idEdge.resize(n + 1);
  }

  void addEdge(int u, int v, int cost, int cap) {
    adj[u].push_back(edges.size());
    edges.push_back({u, v, cost, cap});
    adj[v].push_back(edges.size());
    edges.push_back({v, u, -cost, 0});
  }

  void flow(int src, int sink) {
    maxFlow = 0, minCost = 0;

    while (true) {
      dijkstra(src);
      if (dist[sink] <= -oo || dist[sink] >= oo)
        break;

      int newFlow = oo;
      int cur = sink;
      while (cur != src) {
        newFlow = min(newFlow,
edges[idEdge[cur]].cap);
        cur = par[cur];
      }

      cur = sink;
      while (cur != src) {
        minCost += newFlow *
edges[idEdge[cur]].cost;
        edges[idEdge[cur]].cap -= newFlow;
        edges[idEdge[cur] ^ 1].cap += newFlow;
        cur = par[cur];
      }

      maxFlow += newFlow;
    }
  }

  void dijkstra(int src) {
    for (int i = 1; i <= n; i++)
      dist[i] = oo;
    dist[src] = 0;
    par[src] = -1;

    vector<bool> inQueue(n + 1, 0);
    queue<int> q;
    q.push(src);
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      inQueue[u] = 0;

      for (auto idx: adj[u]) {
        auto [_, v, w, cap] = edges[idx];
        if (cap > 0 && dist[v] > dist[u] + w) {
          dist[v] = dist[u] + w;
          par[v] = u;
          idEdge[v] = idx;
          if (!inQueue[v]) {
            inQueue[v] = 1;
            q.push(v);
          }
        }
      }
    }
  }
}
```

➢ Min Cut

```
struct strongWagner {
  vector<vector<int>> adj;
  vector<int> vis, weight, contract;
  int n;

  strongWagner(int nn) {
    n = nn;
    adj.assign(n + 2, {});
    vis.assign(n + 2, {});
    weight.assign(n + 2, {});
    contract.assign(n + 2, {});
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
        adj[i][j] = 0;
  }

  void addEdge(int x, int y, int v) {
    adj[x][y] += v, adj[y][x] += v;
  }

  int randomMinCut(int &s, int &t) {
    int ret = 0;
    s = t = -1;
    for (int i = 1; i <= n; i++)
      vis[i] = weight[i] = 0;
    for (int i = 1; i <= n; i++) {
      int mx = -oo, u = t;

      for (int j = 1; j <= n; j++)
        if (!contract[j] && !vis[j] && weight[j] > mx)
          mx = weight[j], u = j;

      if (u == t) break;
      s = t, t = u, ret = mx;
      vis[u] = 1;

      for (int j = 1; j <= n; j++)
        if (!contract[j] && !vis[j])
          weight[j] += adj[u][j];
    }
    return ret;
  }

  int minCut() {
    for (int i = 1; i <= n; i++)
      contract[i] = 0;
    int ret = oo;

    for (int i = 1; i < n; i++) {
      int s, t;
      int stMinCut = randomMinCut(s, t);
      ret = min(ret, stMinCut);
      if (ret == 0) return ret;

      contract[t] = 1;
      for (int j = 1; j <= n; j++) {
        if (!contract[j]) {
          adj[s][j] += adj[t][j];
          adj[j][s] += adj[j][t];
        }
      }
    }
    return ret;
  }
};
```

➢ Tarjan (SCC)

```
struct SCCTarjan {
  vector<vector<int>> adj, comps;
  vector<int> inStack, lowLink, compId, dfn;
  stack<int> st;
  int dfnCnt, n;

  SCCTarjan(int nn) {
    adj.resize(nn + 1);
    inStack.resize(nn + 1);
    lowLink.resize(nn + 1);
    dfn.resize(nn + 1, -1);
    compId.resize(nn + 1);
    n = nn;
    dfnCnt = 1;
  }

  void build() {
    for (int i = 1; i <= n; i++)
      if (dfn[i] == -1)
        tarjan(i);
  }

  void tarjan(int u) {
    lowLink[u] = dfn[u] = dfnCnt++;
    inStack[u] = 1;
    st.push(u);

    for (auto v: adj[u]) {
      if (dfn[v] == -1) {
        tarjan(v);
        lowLink[u] = min(lowLink[u], lowLink[v]);
      } else if (inStack[v]) {
        lowLink[u] = min(lowLink[u], dfn[v]);
      }
    }

    if (lowLink[u] == dfn[u]) {
      comps.push_back({});
      int cur = -1;
      while (cur != u) {
        cur = st.top(), st.pop(), inStack[cur] = 0;
        comps.back().push_back(cur);
        compId[cur] = comps.size() - 1;
      }
    }
  }

  vector<int> outDeg, inDeg;
  vector<vector<int>> dagList;

  void computeCompGraph() {
    int compCnt = comps.size();
    int srcCnt = comps.size(), desCnt = comps.size();

    outDeg.resize(compCnt);
    inDeg.resize(compCnt);
    dagList.resize(compCnt);

    for (int u = 1; u <= n; u++) {
      for (auto v: adj[u]) {
        if (compId[u] != compId[v]) {
          dagList[compId[u]].push_back(compId[v]);
          if (!inDeg[compId[v]]++) srcCnt--;
          if (!outDeg[compId[u]]++) desCnt--;
        }
      }
    }

    /* Min edges to convert DAG to one cycle */
    int cnt = max(srcCnt, desCnt);
    if (comps.size() == 1)
      cnt = 0;
    cout << cnt;
  }
};
```

➢ Edges to Make Dag SCC

```
vector<int> vis, id;
int tc = 1;

void dfs(int u, SCCTarjan &scc) {
  vis[u] = tc;

  for (auto v: scc.dagList[u]) {
    if (!vis[v]) dfs(v, scc);
    id[tc] = min(id[vis[v]], id[tc]);
  }
}

void ROOM() {
  int n;
  cin >> n;
  vis.assign(n + 1, 0);
  id.assign(n + 1, 0);
  SCCTarjan scc(n);
  for (int i = 0; i < n; i++) {
    int u;
    cin >> u;
    scc.adj[i + 1].push_back(u);
  }
  scc.build();
  scc.computeCompGraph();
  if (scc.comps.size() == 1)
    return;

  id[tc] = 1;
  for (int i = 0; i < scc.comps.size(); i++)
    if (!vis[i])
      dfs(i, scc), tc++, id[tc] = tc;

  vector<int> can(n + 1);
  vector<int> v;
  for (int i = 0; i < scc.comps.size(); i++) {
    int cur = id[vis[i]];
    if (!can[cur])
      v.push_back(cur);
    can[cur] = 1;
  }
  sort(v.begin(), v.end());

  vector<int> inDags(n + 1, -1), outDags(n + 1, -1);
  vector<int> outs, ins;
  for (int u = 1; u <= n; u++) {
    int i = scc.compId[u];
    int cur = id[vis[i]];
    if (!scc.inDeg[i] && inDags[cur] == -1)
      inDags[cur] = u;
    else if (!scc.inDeg[i])
      ins.push_back(u);
    scc.inDeg[i]++;

    if (!scc.outDeg[i] && outDags[cur] == -1)
      outDags[cur] = u;
    else if (!scc.outDeg[i])
      outs.push_back(u);
    scc.outDeg[i]++;
  }

  for (int i = 0; i < v.size() - 1; i++) {
    int cur1 = v[i], cur2 = v[i + 1];
    cout << outDags[cur1] << " " << inDags[cur2] <<
"\n";
  }

  int cur1 = v.back(), cur2 = v[0];
  cout << outDags[cur1] << " " << inDags[cur2] <<
"\n";


  if (outs.empty() && ins.empty())
    return;
  if (outs.empty())
    outs.push_back(outDags[1]);
  if (ins.empty())
    ins.push_back(inDags[1]);

  while (outs.size() && ins.size()) {
    cout << outs.back() << " " << ins.back() << "\n";
    if (outs.size() == 1 && ins.size() == 1)
      break;
    if (ins.size() != 1)
      ins.pop_back();
    if (outs.size() != 1)
      outs.pop_back();
  }
}
```

➢ Tarjan Bridges

```
struct bridgesTarjan {
  vector<vector<int>> adj;
  vector<pair<int, int>> bridges;
  vector<int> lowLink, dfn;
  int dfnCnt, n;

  bridgesTarjan(int nn) {
    adj.resize(nn + 1);
    lowLink.resize(nn + 1);
    dfn.resize(nn + 1, -1);
    n = nn;
    dfnCnt = 1;
  }

  void build() {
    for (int i = 1; i <= n; i++)
      if (dfn[i] == -1)
        tarjan(i);

    sort(bridges.begin(), bridges.end());
  }

  void tarjan(int u, int par = -1) {
    lowLink[u] = dfn[u] = dfnCnt++;

    for (auto v: adj[u]) {
      if (v == par)
        continue;

      if (dfn[v] == -1) {
        tarjan(v, u);
        lowLink[u] = min(lowLink[u], lowLink[v]);
        if (lowLink[v] == dfn[v])
          bridges.push_back({min(u, v), max(u, v)});
      } else {
        lowLink[u] = min(lowLink[u], dfn[v]);
      }
    }
  }
};
```

➢ Tarjan Art

```
struct artPointTarjan {
  vector<vector<int>> adj;
  vector<vector<pair<int, int>>> BCC;
  vector<int> isArtPoint, lowLink, dfn;
  stack<pair<int, int>> st;
  int dfnCnt, n;
  bool root;

  artPointTarjan(int nn) {
    adj.resize(nn + 1);
    lowLink.resize(nn + 1);
    dfn.resize(nn + 1, -1);
    isArtPoint.resize(nn + 1);
    n = nn;
    dfnCnt = 1;
    root = false;
  }

  void build() {
    for (int i = 1; i <= n; i++)
      if (dfn[i] == -1)
        tarjan(i);
  }
```

```
  void tarjan(int u, int par = -1) {
    lowLink[u] = dfn[u] = dfnCnt++;
    pair<int, int> edge;

    for (auto v: adj[u]) {
      if (par != v && dfn[v] < dfn[u])
        st.push({u, v});

      if (dfn[v] == -1) {
        tarjan(v, u);
        lowLink[u] = min(lowLink[u], lowLink[v]);

        if (lowLink[v] >= dfn[u]) {
          if (dfn[u] == 1 && root == false)
            root = true;
          else
            isArtPoint[u] = 1;

          int cnt = 0;
          BCC.push_back({});
          do {
            cnt++;
            edge = st.top();
            st.pop();
            BCC.back().push_back(edge);
          } while (edge.first != u || edge.second != v);
          if (cnt == 1)
            BCC.back().push_back(edge);
        }
      } else if (v != par)
        lowLink[u] = min(lowLink[u], dfn[v]);
    }
  }
};
```

➢ 2 - SAT

```
struct twoSat {
  vector<int> assignedValue;
  int n;

  twoSat(int nn) {
    n = 2 * nn;
    assignedValue.resize(n + 1, -1);
  }

  int NOT(int u) { return ((1 ^ (u - 1)) + 1); }

  // (0, 1), (1, 0), (1, 1)
  void addEitherOrBoth(int u, int v, SCCTarjan &SCC)
{
    SCC.adj[NOT(u)].push_back(v);
    SCC.adj[NOT(v)].push_back(u);
  }

  // (0, 1), (1, 0)
  void addEitherNotBoth(int u, int v, SCCTarjan
&SCC) {
    addEitherOrBoth(u, v, SCC);
    addEitherOrBoth(NOT(u), NOT(v), SCC);
  }

  // (1, 1, 1, 1, 1, 0) || (1, 1, 1, 1, 1, 1)
  void addAllExpectAtMostOne(vector<int> &v,
SCCTarjan &SCC){
    cnt += 2;
    addEitherOrBoth(v[0], NOT(cnt), SCC);
    for(int i = 1; i < v.size(); i++){
      cnt += 2;
      addEitherOrBoth(v[i], NOT(cnt), SCC);
      addEitherOrBoth(v[i], cnt - 2, SCC);
      addEitherOrBoth(cnt - 2, NOT(cnt), SCC);
    }
  }


  // (0, 0, 0, 0, 0, 1) || (0, 0, 0, 0, 0, 0)
  void addAtMostOne(vector<int> &v, SCCTarjan
&SCC){
    cnt += 2;
    addEitherOrBoth(NOT(v[0]), cnt, SCC);
```

```
    for(int i = 1; i < v.size(); i++){
      cnt += 2;
      addEitherOrBoth(NOT(v[i]), cnt, SCC);
      addEitherOrBoth(NOT(v[i]), NOT(cnt - 2), SCC);
      addEitherOrBoth(NOT(cnt - 2), cnt, SCC);
    }
  }

  void forceValue(int u, bool f, SCCTarjan &SCC) {
    if (f)
      SCC.adj[NOT(u)].push_back(u);
    else
      SCC.adj[u].push_back(NOT(u));
  }

  void remove(int u, bool f, SCCTarjan &scc){
    if (f)
      scc.adj[NOT(u)].pop_back();
    else
      scc.adj[u].pop_back();
  }

  bool isSolvable(SCCTarjan &SCC) {
    for (int u = 1; u <= n; u += 2)
      if (SCC.compId[u] == SCC.compId[NOT(u)])
        return false;
    return true;
  }

  void assignValues(SCCTarjan &SCC) {
    vector<int>
compAssignedValue(SCC.comps.size(), -1);

    for (int i = 0; i < SCC.comps.size(); i++) {
      if (compAssignedValue[i] == -1) {
        compAssignedValue[i] = 1;
        int notComp =
SCC.compId[NOT(SCC.compRootNode[i])];
        compAssignedValue[notComp] = 0;
      }
    }

    for (int i = 1; i <= n; i++)
      assignedValue[i] =
compAssignedValue[SCC.compId[i]];
  }

};
```

```
/*
 * take u and v from the input
 * u = 2 * u - 1
 * v = 2 * v - 1
 * use them in 2-sat
 * cnt = 2 * n - 1
 * */
```

➢ LCA (Binary Lifting)

```
struct LCA {
  const int ign = -2e9; // change this
  int n, lg;
  vector<int> depth;
  vector<vector<int>> up, qu;
  vector<vector<pair<int, int>>> adj;

  LCA(vector<vector<pair<int, int>>> &_adj, int root = 1) {
    adj = _adj;
    n = int(adj.size());
    lg = __lg(n) + 1;
    depth = vector<int>(n + 2);
    up = qu = vector<vector<int>>(n + 2, vector<int>(lg + 1));

    dfs(root, 0);
  }

  void dfs(int u, int p) {
    up[u][0] = p;

    for (int i = 1; i <= lg; ++i) {
      up[u][i] = up[up[u][i - 1]][i - 1];
      qu[u][i] = max(qu[u][i - 1], qu[up[u][i - 1]][i - 1]);
    }

    for (auto &[ch, w]: adj[u]) {
      if (ch == p)
        continue;

      depth[ch] = depth[u] + 1, qu[ch][0] = w;
      dfs(ch, u);
    }
  }

  int get_kth_ancestor(int a, int k) {
    if (depth[a] < k)
      return -1;

    for (int i = lg; i >= 0; --i) {
      if (k >= (1 << i)) {
        a = up[a][i];
        k -= (1 << i);
      }
    }

    return a;
  }

  int get(int a, int dist) {
    if (depth[a] < dist or dist == -1)
      return ign;

    int ret = ign;
    for (int i = lg; i >= 0; --i) {
      if (dist >= (1 << i)) {
        ret = max(ret, qu[a][i]);
        a = up[a][i];
        dist -= (1 << i);
      }
    }

    ret = max(ret, qu[a][0]);
    return ret;
  }

  int get_max(int a, int b) {
    int c = get_lca(a, b);
    return max(get(a, distance(a, c) - 1), get(b, distance(b, c) - 1));
  }

  int get_lca(int a, int b) {
    if (depth[a] < depth[b])
      swap(a, b);

    int k = depth[a] - depth[b];
    for (int i = lg; i >= 0; --i)
      if (k & (1 << i))
        a = up[a][i];

    if (a == b) return a;

    for (int i = lg; i >= 0; --i) {
      if (up[a][i] != up[b][i]) {
        a = up[a][i];
        b = up[b][i];
      }
    }

    return up[a][0];
  }

  int distance(int a, int b) {
    int c = get_lca(a, b);
    int x = depth[a] - depth[c];
    int y = depth[b] - depth[c];
    return x + y;
  }
};
```

➢ LCA (Euler)

```
struct LCA {
  int n;
  vector<pair<int, int>> tour;
  vector<int> depth, in, par;
  SparseTable<pair<int, int>> table;

  LCA(int sz) {
    n = sz;
    in.resize(n);
    depth.resize(n);
    par.resize(n);
    table = SparseTable<pair<int, int>>(2 * n);
  }

  void dfs(int u, int p, int d, auto &adj) {
    in[u] = (int) tour.size();
    depth[u] = d;
    par[u] = p;
    tour.emplace_back(d, u);
    for (int v: adj[u]) {
      if (v == p) continue;
      dfs(v, u, d + 1, adj);
      tour.emplace_back(d, u);
    }
  }

  void build(auto &adj) {
    dfs(0, 0, 0, adj);
    table.build(tour);
  }

  int get(int u, int v) {
    return table.query(min(in[u], in[v]), max(in[u], in[v])).second;
  }

  int dist(int u, int v) {
    return depth[u] + depth[v] - 2 * depth[get(u, v)];
  }
};
```

➢ HLD

```
vector<vector<int>> adj;
vector<int> par, depth, pos, sz, heavy, head, val;
int cnt = 0;

void dfs(int u, int p, int d = 0) {
  par[u] = p;
  depth[u] = d;
  sz[u] = 1;

  int mx = 0;
  for (int &v: adj[u]) {
    if (v == p) continue;
    dfs(v, u, d + 1);
    if (sz[v] > mx) heavy[u] = v, mx = sz[v];
    sz[u] += sz[v];
  }
}

void HLD(int u, int hd) {
  if (!u) return;
  head[u] = hd;

  pos[u] = cnt++;
  HLD(heavy[u], hd);
  for (int &v: adj[u])
    if (v != par[u] && v != heavy[u])
      HLD(v, v);
}

int get_path(int u, int v) {
  int a = head[u], b = head[v];

  int res = 0;

  while (a != b) {
    if (depth[a] < depth[b])
      swap(a, b), swap(u, v);
    res = res + query(pos[a], pos[u]);
    u = par[a], a = head[u];
  }

  if (pos[u] > pos[v]) swap(u, v);
  res = res + query(pos[u], pos[v]);
  return res;
}

int get_subtree(int u) {
  return query(pos[u], pos[u] + sz[u]);
}

// one indexed
// call init(n), dfs(1, 0), HLD(1, 1); in main
void init(int n) {
  ++n, cnt = 0;
  adj.assign(n, {});
  par.assign(n, {});
  depth.assign(n, {});
  pos.assign(n, {});
  sz.assign(n, {});
  heavy.assign(n, {});
  head.assign(n, {});
  val.assign(n, {});
}
```

➢ Segment Tree

```
struct Node {
  int ign = 0, val;

  Node() : val(ign) {};

  Node(int x) : val(x) {};

  void set(int x) {
    val = x;
  }
};

#define lNode (x * 2 + 1)
#define rNode (x * 2 + 2)
#define md (lx + (rx - lx) / 2)

struct Sagara {
  int n;
  vector<Node> node;

  Sagara(int sz) {
    n = 1;
    while (n < sz) n *= 2;
    node.assign(n * 2, Node());
  }

  Node merge(Node &l, Node &r) {
    Node res = Node();
    res.val = l.val + r.val;
    return res;
  }

  void build(vector<int> &v, int x, int lx, int rx) {
    if (rx - lx == 1) {
      if (lx < v.size()) node[x] = Node(v[lx]);
      return;
    }

    build(v, lNode, lx, md);
    build(v, rNode, md, rx);
    node[x] = merge(node[lNode], node[rNode]);
  }

  void build(vector<int> &v) { build(v, 0, 0, n); }

  void set(int &ind, ll &val, int x, int lx, int rx) {
    if (rx - lx == 1) return node[x].set(val);

    if (ind < md) set(ind, val, lNode, lx, md);
    else set(ind, val, rNode, md, rx);

    node[x] = merge(node[lNode], node[rNode]);
  }

  void set(int ind, ll val) { set(ind, val, 0, 0, n); }

  Node query(int &l, int &r, int x, int lx, int rx) {
    if (lx >= r || rx <= l) return Node();
    if (rx <= r && lx >= l) return node[x];

    Node L = query(l, r, lNode, lx, md);
    Node R = query(l, r, rNode, md, rx);

    return merge(L, R);
  }

  Node query(int l, int r) {
    return query(l, r, 0, 0, n);
  }
};
```

➢ Lazy Segment Tree

```
struct Node {
  int ign = 0, lazy = 0, val = ign;
  bool isLazy = 0;

  Node() {}

  Node(ll x) : val(x) {}

  void add(int x, int lx, int rx) {
    val += x * (rx - lx);
    lazy += x;
    isLazy = 1;
  }
};

#define lNode (x * 2 + 1)
#define rNode (x * 2 + 2)
#define md (lx + (rx - lx) / 2)

struct Sagara {
  int n;
  vector<Node> node;

  Sagara(int sz) {
    n = 1;
    while (n < sz) n *= 2;
    node.assign(n * 2, Node());
  }

  Node merge(Node &l, Node &r) {
    Node res = Node();
    res.val = l.val + r.val;
    return res;
  }

  void propagate(int x, int lx, int rx) {
    if (rx - lx == 1 || !node[x].isLazy) return;

    node[lNode].add(node[x].lazy, lx, md);
    node[rNode].add(node[x].lazy, md, rx);

    node[x].isLazy = node[x].lazy = 0;
  }

  void update(int l, int r, ll val, int x, int lx, int rx) {
    propagate(x, lx, rx);

    if (lx >= r || rx <= l) return;
    if (lx >= l && rx <= r)
      return node[x].add(val, lx, rx);

    update(l, r, val, lNode, lx, md);
    update(l, r, val, rNode, md, rx);

    node[x] = merge(node[lNode], node[rNode]);
  }

  void update(int l, int r, ll val) { update(l, r, val, 0, 0,
n); }

  Node query(int l, int r, int x, int lx, int rx) {
    propagate(x, lx, rx);

    if (lx >= l && rx <= r) return node[x];
    if (lx >= r || rx <= l) return Node();

    Node L = query(l, r, lNode, lx, md);
    Node R = query(l, r, rNode, md, rx);

    return merge(L, R);
  }

  Node query(int l, int r) {
    return query(l, r, 0, 0, n);
  }
};
```

➢ Merge Sort Tree

```
struct Node {
  vector<int> v;
  Node() {};
  Node(int x) : v({x}) {};
};

#define lNode (x * 2 + 1)
#define rNode (x * 2 + 2)
#define md (lx + (rx - lx) / 2)

struct MergeSortSagara {
  vector<Node> node;
  int n;

  MergeSortSagara(int _n) {
    n = 1;
    while (n < _n) n <<= 1;
    node.assign(n * 2, Node());
  }

  Node merge(Node &l, Node &r) {
    Node res;

    int i = 0, j = 0;
    while (i < l.v.size() && j < r.v.size()) {
      if (l.v[i] < r.v[j]) res.v.push_back(l.v[i++]);
      else res.v.push_back(r.v[j++]);
    }

    while (i < l.v.size()) res.v.push_back(l.v[i++]);
    while (j < r.v.size()) res.v.push_back(r.v[j++]);
    return res;
  }

  void build(vector<int> &v, int x, int lx, int rx) {
    if (rx - lx == 1) {
      if (lx < v.size()) node[x] = Node(v[lx]);
      return;
    }

    build(v, lNode, lx, md);
    build(v, rNode, md, rx);
    node[x] = merge(node[lNode], node[rNode]);
  }

  void build(vector<int> &v) { build(v, 0, 0, n); }

  int query(int l, int r, int x, int lx, int rx, int val) {
    if (rx <= l || lx >= r) return 0;
    if (lx >= l && rx <= r) return calc(node[x], val);
    return query(l, r, lNode, lx, md, val) + query(l, r,
rNode, md, rx, val);

  }

  int query(int l, int r, int val) {
    return query(l, r, 0, 0, n, val);
  }

  // change this function to match your need
  int calc(Node &no, int val) {
    return greater_than(no, val);
  }

  int less_than(Node &no, int val) {
    return lower_bound(no.v.begin(), no.v.end(), val) -
no.v.begin();
  }

  int greater_than(Node &no, int val) {
    return no.v.size() - less_than(no, val) - equal(no,
val);
  }

  int equal(Node &no, int val) {
    return upper_bound(no.v.begin(), no.v.end(), val)
- lower_bound(no.v.begin(), no.v.end(), val);
  }
};
```

➢ 2D Segment Tree

```
struct Node {
  ll ign = 0, val;
  Node() : val(ign) {}
  Node(ll x) : val(x) {}
  Node operator+(Node &r) {
    Node res = Node();
    res.val = val + r.val;
    return res;
  }
};

#define lNode (x * 2 + 1)
#define rNode (x * 2 + 2)
#define md (lx + (rx - lx) / 2)

struct Sagara {
  int n;
  vector<Node> node;

  Sagara(int sz) {
    n = 1;
    while (n < sz) n *= 2;
    node.assign(n * 2, Node());
  }

  void set(int &ind, ll &val, int x, int lx, int rx) {
    if (rx - lx == 1) {
      node[x] = Node(val);
      return;
    }

    if (ind < md) set(ind, val, lNode, lx, md);
    else set(ind, val, rNode, md, rx);

    node[x] = node[lNode] + node[rNode];
  }

  void set(int ind, ll val) {set(ind, val, 0, 0, n);}

  Node get(int &i) {return node[n + i - 1];}

  Node query(int &l, int &r, int x, int lx, int rx) {
    if (lx >= r || rx <= l) return Node();
    if (rx <= r && lx >= l) return node[x];
    Node L = query(l, r, lNode, lx, md);
    Node R = query(l, r, rNode, md, rx);
    return L + R;
  }

  Node query(int l, int r) {
    return query(l, r, 0, 0, n);
  }
};

struct Sagara2D {
  vector<Sagara> node;
  int n;

  Sagara2D(int _n, int _m) {
    n = 1;
    while (n < _n) n <<= 1;
    node.assign(n * 2, Sagara(_m));
  }

  void set(int i, int j, int val, int x, int lx, int rx) {
    if (rx - lx == 1)
      return node[x].set(j, val);

    if (i < md) set(i, j, val, lNode, lx, md);
    else set(i, j, val, rNode, md, rx);

    Node L = node[lNode].get(j);
    Node R = node[rNode].get(j);

    node[x].set(j, (L + R).val);
  }

  void set(int i, int j, int val) {
    set(i, j, val, 0, 0, n);
  }

  // r and b are not included
  int query(int t, int b, int l, int r, int x, int lx, int rx) {
    if (rx <= t || lx >= b) return 0;
    if (lx >= t && rx <= b)
      return node[x].query(l, r).val;

    int L = query(t, b, l, r, lNode, lx, md);
    int R = query(t, b, l, r, rNode, md, rx);
    return L + R;
  }
  // top, bottom, right, left
  int query(int t, int b, int r, int l) {
    return query(t, b, r, l, 0, 0, n);
  }
};
```

➢ PST

```
struct PST {
  // root[query_index] returns the root of the version
of this query
  // L[x], R[x] are the id's of x left and right child
  vector<int> node, L, R, root;
  int n, curr;
  const int ign = 0; // change this accordigly

  PST(int _n, int q) {
    const int LOG = 30;
    n = _n;
    root.reserve(q + 10);
    L.resize(n * LOG);
    R.resize(n * LOG);
    node.resize(n * LOG);
    curr = 0;
  }

  int merge(int lf, int ri) {
    return lf + ri;
  }

  int init(int lx, int rx) {
    int id = curr++;
    if (rx - lx < 2) {
      node[id] = ign;
      return id;
    }

    int md = (lx + rx) / 2;
    L[id] = init(lx, md);
    R[id] = init(md, rx);

    node[id] = merge(node[L[id]], node[R[id]]);
    return id;
  }

  void init() {
    root.push_back(init(0, n));
  }

  int set(int ind, int val, int x, int lx, int rx) {
    if (ind >= rx || ind < lx) return x;

    int id = curr++;
    if (rx - lx < 2) {
      node[id] = val;
      return id;
    }

    int md = (rx + lx) / 2;
    L[id] = set(ind, val, L[x], lx, md);
    R[id] = set(ind, val, R[x], md, rx);

    node[id] = merge(node[L[id]], node[R[id]]);
    return id;
  }

  void set(int ind, int val, int version = -1) {
    if (~version)
      root.push_back(set(ind, val, root[version], 0,
n));
    else
      root.push_back(set(ind, val, root.back(), 0, n));
// work on the latest version
  }

  int query(int l, int r, int x, int lx, int rx) {
    if (rx <= l || lx >= r) return ign;
    if (rx <= r && lx >= l) return node[x];
    int md = (lx + rx) / 2;
    return merge(query(l, r, L[x], lx, md), query(l, r,
R[x], md, rx));
  }

  int query(int l, int r, int version = -1) {
    if (~version)
      return query(l, r, root[version], 0, n);
    else
      return query(l, r, root.back(), 0, n);
  }
};
```

➢ Segment Tree Functions

```
// seg tree with hash remember to multiply with inv[l] in the query

// to get prev go to the right before the left

int next_greater(int l, int r, int val, int x, int lx, int rx) {
  if (lx >= r or l >= rx) return -1;
  if (node[x].mx <= val) return -1;
  if (rx - lx == 1) return lx;

  int ans = next_greater(l, r, val, lNode, lx, md);

  if (ans == -1)
    return next_greater(l, r, val, rNode, md, rx);
  return ans;
}

int next_smaller(int l, int r, int val, int x, int lx, int rx) {
  if (lx >= r or l >= rx) return -1;
  if (node[x].mn >= val) return -1;
  if (rx - lx == 1) return lx;

  int ans = next_smaller(l, r, val, lNode, lx, md);

  if (ans == -1)
    return next_smaller(l, r, val, rNode, md, rx);
  return ans;
}

int kth_one(int k, int x, int lx, int rx) {
  if (rx - lx == 1)
    return lx;
  if (k < node[lNode].sum)
    return find_kth_one(k, lNode, lx, md);
  return find_kth_one(k - node[lNode].sum, rNode, md, rx);
}

int kth_one(int k) {
  if(node[0].sum < k) return -1;
  return kth_one(k, 0, 0, n);
}
```

```
// max subarray sum

Node merge(Node &l, Node &r) {
  Node res = Node();
  res.ans = max({l.ans, r.ans, l.suff + r.pre});
  res.pre = max(l.pre, l.sum + r.pre);
  res.suff = max(r.suff, r.sum + l.suff);
  res.sum = r.sum + l.sum;
  return res;
}
```

➢ Iterative Segment Tree

```
const int N = 2e5 + 1;  // limit for array size
int t[2 * N];
int m;  // array size

void build(vector<int> &v) {  // build the tree
  for (int i = 0; i < m; ++i) t[i + m] = v[i];
  for (int i = m - 1; i > 0; --i)
    t[i] = t[i << 1] + t[i << 1 | 1];
}

void modify(int p, int value) {
  for (t[p += m] = value; p > 1; p >>= 1)
    t[p >> 1] = t[p] + t[p ^ 1];
}

int query(int l, int r) {  // sum on interval [l, r)
  int res = 0;
  for (l += m, r += m; l < r; l >>= 1, r >>= 1) {
    if (l % 2) res += t[l++];
    if (r % 2) res += t[--r];
  }
  return res;
}
```

➢ Sparse Table

```
template<class T>
struct SparseTable {
  // change mxLog
  static const int mxLog = 21;
  vector<array<T, mxLog>> table;
  vector<int> lg;
  int n;

  SparseTable(int sz) {
    n = sz;
    table.resize(n + 1);
    lg.resize(n + 1);
    for (int i = 0; i <= n; ++i) lg[i] = __lg(i);
  }

  void build(vector<T> &v) {
    for (int i = 0; i < n; ++i) table[i][0] = v[i];
    for (int j = 1; j < mxLog; ++j)
      for (int i = 0; i + (1 << j) - 1 < n; ++i)
        table[i][j] = merge(table[i][j - 1], table[i + (1 <<
(j - 1))][j - 1]);
  }

  T merge(T &l, T &r) { return min(l, r); }

  T query(int l, int r) {
    int j = lg[r - l + 1];
    return merge(table[l][j], table[r - (1 << j) + 1][j]);
  }
};
```

➢ BIT

```
struct FenwickTree {
  vector<ll> bit;
  int n;

  void updateBit(int ind, ll val) {
    for (; ind < n; ind = ind | (ind + 1))
      bit[ind] += val;
  }

  FenwickTree(int _n) {
    n = _n + 1;
    bit.assign(n, 0);
  }

  void updateRange(int l, int r, ll val) {
    updateBit(l, val);
    updateBit(r + 1, -val);
  }

  ll query(int r) {
    ll ret = 0;
    for (; r >= 0; r = (r & (r + 1)) - 1)
      ret += bit[r];
    return ret;
  }

  ll prefix(int l, int r) { return query(r) - query(l - 1); }
};
```

➢ Ordered Set

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;

void myErase(ordered_set<int> &t, int v) {
  int rank = t.order_of_key(v);
  ordered_set<int>::iterator it = t.find_by_order(rank);
  t.erase(it);
}
```

➢ Convex Hull

```
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line &o) const { return k < o.k;
}
  bool operator<(ll x) const { return p < x; }
};

struct HullDynamic : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  const ll inf = 2e18;

  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }

  bool isect(iterator x, iterator y) {
    if (y == end()) {
      x->p = inf;
      return false;
    }
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }

  void add(ll k, ll m) { // k is the slope
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y =
erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }

  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

➢ Matrix

```
template<class T>
struct Matrix {
  int n, m;
  vector<vector<T>> a;

  Matrix(int _n, int _m) {
    n = _n, m = _m;
    a.assign(n, vector<T>(m));
  }

  Matrix operator*(const Matrix &b) {
    int r = n, c = b.m, k = m;
    Matrix res(r, c);
    for (int i = 0; i < r; ++i)
      for (int j = 0; j < c; ++j)
        for (int o = 0; o < k; ++o)
          res.a[i][j] = add(res.a[i][j], mul(a[i][o],
b.a[o][j], MOD), MOD);
    return res;
  }


  friend Matrix power(Matrix mat, ll p) {
    Matrix ret(mat.n, mat.n);
    for (int i = 0; i < mat.n; ++i)
      ret.a[i][i] = 1;

    while (p) {
      if (p & 1) ret = ret * mat;
      mat = mat * mat, p >>= 1;
    }
    return ret;
  }
};
```

➢ BIT 2D

```
// this supports update rect and get a single cell
struct BIT2D {
  int n, m;
  vector<vector<int>> bit;

  BIT2D(int _n, int _m) {
    n = _n + 5, m = _m + 5;
    bit.assign(n, vector<int>(m));
  }

  void updateY(int x, int ind, ll val) {
    for (; ind < m; ind |= (ind + 1))
      bit[x][ind] += val;
  }

  void update(int x, int y, int val) {
    for (; x < n; x |= (x + 1))
      updateY(x, y, val);
  }

  // pass it left, right, top, bottom, value
  void updateRect(int l, int r, int t, int b, ll val) {
    update(t, l, val);
    update(t, r + 1, -val);
    update(b + 1, l, -val);
    update(b + 1, r + 1, val);
  }

  int getY(int x, int ind) {
    int v = 0;
    for (; ind >= 0; ind = (ind & (ind + 1)) - 1)
      v += bit[x][ind];
    return v;
  }

  int get(int x, int y) {
    int v = 0;
    for (; x >= 0; x = (x & (x + 1)) - 1)
      v += getY(x, y);
    return v;
  }
};
```

➢ MO Algorithm

```
struct Query {
  int l, r, ind;
};

struct MO {
  int n, sq, l, r;
  ll curr;
  vector<int> v;

  MO(vector<int> &_v) {
    v = _v;
    n = v.size();
    sq = sqrt(n) + 1;
    curr = 0;
  };

  void add(int i) {}
  void del(int i) {}

  void move(int &lq, int &rq) {
    while (r < rq) add(++r);
    while (l < lq) del(l++);
    while (l > lq) add(--l);
    while (r > rq) del(r--);
  }

  void solve(vector<Query> &qu) {
    sort(qu.begin(), qu.end(), [&](auto &lf, auto ri) {
      if (lf.l / sq == ri.l / sq)
        return lf.r < ri.r;
      return lf.l / sq < ri.l / sq;
    });

    l = qu[0].l, r = qu[0].l;
    add(l);

    vector<int> res(qu.size());
    for (auto &[lq, rq, iq]: qu) {
      move(lq, rq);
      res[iq] = curr;
    }

    for (int &i: res) cout << i << endl;
  }
};
```

➢ MO with Updates

```
const int N = 2e5 + 5, SQ = 3500;

struct query {
  int l, r, idx, uIdx;

  bool operator<(const query &other) const {
    if (l / SQ != other.l / SQ)
      return l / SQ < other.l / SQ;
    if (r / SQ != other.r / SQ)
      return r / SQ < other.r / SQ;
    return uIdx < other.uIdx;
  }
};

struct update {
  int idx, val, old;
};

int a[N], ans[N], frq[N], cnt[N];

void add(int idx) {}
void remove(int idx) {}

void upd(update &u, int l, int r) {
  if (u.idx >= l && u.idx <= r)remove(u.idx);
  a[u.idx] = u.val;
  if (u.idx >= l && u.idx <= r)add(u.idx);
}

void cancel(update &u, int l, int r) {
  if (u.idx >= l && u.idx <= r)remove(u.idx);
  a[u.idx] = u.old;
  if (u.idx >= l && u.idx <= r)add(u.idx);
}

void mo(vector<query> &v, vector<update> &u) {
  int l = 0, r = -1;
  int cur = 0;
  for (auto &q: v) {
    while (cur < q.uIdx)upd(u[cur++], l, r);
    while (cur > q.uIdx)cancel(u[--cur], l, r);
    while (r < q.r)add(++r);
    while (l > q.l)add(--l);
    while (r > q.r)remove(r--);
    while (l < q.l)remove(l++);
    while (cnt[ans[q.idx]])ans[q.idx]++;
  }
}
```

Bitmasks

```
int count_numbers_has_ith_bit(int n, int k) {
  ++n;
  int d = ( 1LL << (k + 1) ), p = (1LL << k);
  return n / d * p + max(0LL, n % d - p);
}

int highest_bit(int x) {
  int t = 63 - __builtin_clzll(x);
  return (1LL << t);
}

int lowest_bit(int x) {
  int t = __builtin_ctzll(x);
  return (1LL << t);
}

// loop through submasks

for (int j = mask; j; j = (j - 1) & mask) {

}

int xor_range(int n) { // from 1 to n
  if (n % 4 == 0) return n;
  if (n % 4 == 1) return 1;
  if (n % 4 == 2) return n + 1;
  return 0;
}

int odd_xor(int n) {
  if (n % 2 == 0)
    return ((xor_range(n)) ^ (2LL * xor_range(n / 2LL)));
  else
    return ((xor_range(n)) ^ (2LL * xor_range((n - 1LL) /
2LL)));
}

int odd_xor_range(int l, int r) {
  return odd_xor(l - 1) ^ odd_xor(r);
}

int even_xor_range(int l, int r) {
  int xor_r = 2LL * xor_range(r / 2LL);
  int xor_l = 2LL * xor_range((l - 1LL) / 2LL);
  return (xor_l ^ xor_r);
}
```

➢ XOR of all subarrays

```
int calcSubArrayXORSum(vector<int> &arr) {
  int n = arr.size();
  int sum = 0;
  int multiplier = 1;
  for (int i = 0; i < 30; i++) {
    int oddCount = 0;
    bool isOdd = 0;
    for (int j = 0; j < n; j++) {
      if ((arr[j] & (1 << i)) > 0)
        isOdd = (!isOdd);
      if (isOdd)
        oddCount++;
    }
    for (int j = 0; j < n; j++) {
      sum += (multiplier * oddCount);
      if ((arr[j] & (1 << i)) > 0)
        oddCount = (n - j - oddCount);
    }
    multiplier *= 2;
  }
  return sum;
}
```

➢ SOS DP

```
const int B = 20;
const int M = 1 << B;

// subset contribute to its superset
void forward(vector<int> &dp) {
  for (int i = 0; i < B; ++i)
    for (int m = 0; m < M; ++m)
      if (m & (1 << i))
        dp[m] += dp[m ^ (1 << i)];
}

// superset contribute to its subset
void forwardRev(vector<int> &dp) {
  for (int i = 0; i < B; ++i)
    for (int m = M - 1; ~m; --m)
      if (m & (1 << i))
        dp[m ^ (1 << i)] += dp[m];
}

// remove subset contribution from superset
void backward(vector<int> &dp) {
  for (int i = 0; i < B; ++i)
    for (int m = M - 1; ~m; --m)
      if (m & (1 << i))
        dp[m] -= dp[m ^ (1 << i)];
}

// remove superset contribution from subset
void backwardRev(vector<int> &dp) {
  for (int i = 0; i < B; ++i)
    for (int m = 0; m < M; ++m)
      if (m & (1 << i))
        dp[m ^ (1 << i)] -= dp[m];
}
```

➢ XOR of all subarrays multiplied by length

```
int calcSubArrayXORSum(vector<int> arr) {
  int n = arr.size();
  int sum = 0;
  int multiplier = 1;
  for (int i = 0; i < 30; i++) {
    int oddCount = 0;
    bool isOdd = 0;
    int tot = 0;
    for (int j = 0; j < n; j++) {
      if ((arr[j] & (1 << i)) > 0)
        isOdd = (!isOdd);
      if (isOdd) {
        oddCount++;
        tot += j + 1;
      }
    }
    for (int j = 0; j < n; j++) {
      sum += ((multiplier % mod) * (tot % mod)) %
mod;
      sum %= mod;
      if ((arr[j] & (1 << i)) > 0) {
        oddCount = (n - j - oddCount);
        tot = (n - j) * (n - j + 1) / 2 - tot;
        tot -= oddCount;
      } else
        tot -= oddCount;
    }
    multiplier *= 2;
  }
  return sum;
}
```

➢ Suffix Array

```
struct SuffixArray {
  int n;
  vector<int> suff, lcp, c;

  SuffixArray(int sz) {
    n = sz + 1;
    suff.resize(n);
    lcp.resize(n);
    c.resize(n);
  }

  void countingSort(vector<int> &p) {
    vector<int> cnt(n), pos(n), newP(n);
    for (int i: c)
      cnt[i]++;


    for (int i = 1; i < n; ++i)
      pos[i] = pos[i - 1] + cnt[i - 1];


    for (int i: p)
      newP[pos[c[i]]++] = i;
    swap(p, newP);
  }

  void build(string &s) {
    s += '$';
    vector<pair<char, int>> a(n);

    for (int i = 0; i < n; ++i)
      a[i] = {s[i], i};

    sort(a.begin(), a.end());

    vector<int> p(n);
    for (int i = 0; i < n; ++i)
      p[i] = a[i].second;

    for (int i = 1; i < n; ++i)
      c[a[i].second] = c[a[i - 1].second] + (a[i].first !=
a[i - 1].first);

    int k = 0;
    while ((1 << k) < n) {
      int bit = 1 << k;

      for (int i = 0; i < n; ++i)
        p[i] = (p[i] - bit + n) % n;

      countingSort(p);

      vector<int> newC(n);
      for (int i = 1; i < n; ++i) {
        int currL = p[i], currR = (p[i] + bit) % n;
        int preL = p[i - 1], preR = (p[i - 1] + bit) % n;
        bool add = (c[currL] != c[preL]) || (c[currR] !=
c[preR]);
        newC[p[i]] = newC[p[i - 1]] + add;
      }

      c = newC;
      ++k;
    }

    suff = p;

    // Build LCP
    k = 0;
    for (int i = 0; i < n - 1; ++i) {
      int pi = c[i];
      int j = p[pi - 1];
      while (s[i + k] == s[j + k])
        ++k;
      lcp[pi] = k;
      k = max(int(0), k - 1);
    }
  }
};
```

➢ Hashing

```
mt19937
rng(chrono::steady_clock::now().time_since_epoch()
.count());
#define getrand(l, r) uniform_int_distribution<long
long>(l, r)(rng)

struct Hash {
  // everything is zero indexed
  int mod, base, st, N;
  vector<int> pw, inv;

  // st is the start char, _N is the max size of the string
  Hash(int _st = 'a', int _N = 1e6) {
    st = _st - 1;
    N = _N + 1;
    pw.resize(N);
    inv.resize(N);
    gen();
    pre();
  }

  void gen() {
    auto check = [](int x) {
      for (int i = 2; i * i <= x; ++i)
        if (!(x % i)) return false;
      return true;
    };
    mod = getrand(1e8, 2e9);
    base = getrand(30, 120);
    while (!check(mod))--mod;
  }

  void pre() {
    int inv_pw = inverse(base, mod);
    pw[0] = inv[0] = 1;
    for (int i = 1; i < N; ++i) {
      pw[i] = mul(pw[i - 1], base, mod);
      inv[i] = mul(inv[i - 1], inv_pw, mod);
    }
  }
  int push_back(int h, char a, int len) {
    return add(h, mul(a - st, pw[len], mod), mod);
  }

  int push_front(int h, char a) {
    return add(a - st, mul(h, base, mod), mod);
  }

  int concat(int l, int r, int szLeft) {
    return add(l, mul(r, pw[szLeft], mod), mod);
  }

  vector<int> build(string &s) {
    int sz = s.size();
    vector<int> res(sz);
    res[0] = s[0] - st;
    for (int i = 1; i < sz; ++i)
      res[i] = push_back(res[i - 1], s[i], i);
    return res;
  }

  // [l, r] l & r are included
  int getSubstring(int l, int r, vector<int> &hash) {
    int res = hash[r];
    if (l) res = add(res, -hash[l - 1], mod);
    return mul(res, inv[l], mod);
  }

  bool is_palindrome(int l, int r, vector<int> &hsh,
vector<int> &hshRev) {
    int rev_l = hsh.size() - r - 1;
    int rev_r = hsh.size() - l - 1;
    return getSubstring(l, r, hsh) ==
getSubstring(rev_l, rev_r, hshRev);
  }

  // get Hash of path on tree
  void dfs(int u, int p, int l = 0) {
    hsh[u] = h.push_back(hsh[p], c[u], l);
    hshRev[u] = h.push_front(hshRev[p], c[u]);
    lvl[u] = l;
    par[u] = p;

    for (int &v: adj[u])
      if (v != p) dfs(v, u, l + 1);
  }

  int getTreePath(int u, int v, int lc) {
    int u_lc = add(hshRev[u], -mul(hshRev[lc],
pw[lvl[u] - lvl[lc]], mod), mod);

    int lc_v = add(hsh[v], -hsh[par[lc]], mod);
    lc_v = mul(lc_v, inv[lvl[lc]], mod);

    return concat(u_lc, lc_v, lvl[u] - lvl[lc]);
  };
};
```

➢ Trie & Aho-Corasick

```
// mx char
const int MX = 26;

struct Trie {
  struct Node {
    array<int, MX> nxt;
    vector<int> pat;
    int f = 0;

    Node() { nxt.fill(-1); }
  };

  vector<Node> node;
  int start = 'a';

  Trie() {
    node.resize(1);
  }

  void add(string &s, int &i) {
    int v = 0;
    for (char ch: s) {
      int c = ch - start;
      if (node[v].nxt[c] == -1) {
        node[v].nxt[c] = node.size();
        node.emplace_back();
      }
      v = node[v].nxt[c];
    }

    node[v].pat.push_back(i);
  }
};
```

```
struct AhoCorasick : public Trie {

  AhoCorasick(vector<string> &pat) {
    for (int i = 0; i < pat.size(); ++i)
      add(pat[i], i);
    build();
  }

  int move(int &curr, int &c) {
    int f = node[curr].f;
    while (node[f].nxt[c] == -1 && f) f = node[f].f;
    return node[f].nxt[c] == -1 ? 0 : node[f].nxt[c];
  }

  void build() {
    queue<int> q;

    for (int i = 0; i < MX; i++) {
      if (~node[0].nxt[i])
        q.push(node[0].nxt[i]);
    }

    while (!q.empty()) {
      int curr = q.front();
      q.pop();

      for (int i = 0; i < MX; i++) {
        int nxt = node[curr].nxt[i];
        if (nxt == -1) continue;

        int f = move(curr, i);
        node[nxt].f = f;

        for (auto &j: node[f].pat)
          node[nxt].pat.push_back(j);

        q.push(nxt);
      }
    }
  }
};
```

➢ Manacher

```
struct manacher {
  string s;
  vector<int> p;

  manacher(string &in) {
    s = manacher_string(in);

    int n = s.size();
    p.assign(n, 0);
    for (int i = 0, l = 0, r = -1, k; i < n; ++i) {
      if (i > r)
        k = 1;
      else
        k = min(p[l + (r - i)], r - i) + 1;

      while (i - k >= 0 && i + k < n && s[i - k] == s[i + k])
        ++k;

      if (i - k == -1 || i + k == n || s[i - k] != s[i + k])
        --k;

      p[i] = k;
      if (i + k > r)
        l = i - k, r = i + k;
    }
  }

  string manacher_string(string &in) {
    int n = in.size();
    string t(2 * n + 1, '#');
    for (int i = 1, j = 0; i < 2 * n + 1; i += 2, ++j)
      t[i] = in[j];
    return t;
  }

  bool is_palindrome(int l, int r) {
    int L = 2 * l + 1;
    int R = 2 * r + 1;
    int mid = (L + R) / 2;
    return p[mid] >= mid - L;
  }
```

➢ Z Algorithm

```
vector<int> get_z_table(string &txt) {
  int n = int(txt.size());

  int l = 0, r = 0, k;
  vector<int> z_table(n);
  for (int i = 1; i < n; ++i) {
    if (i > r) {
      l = r = i;
      while (r < n and txt[r - l] == txt[r]) {
        r++;
      }
      z_table[i] = r - l, --r;
    } else {
      k = i - l;
      if (z_table[k] < r - i + 1) {
        z_table[i] = z_table[k];
      } else {
        l = i;
        while (r < n and txt[r - l] == txt[r]) {
          ++r;
        }
        z_table[i] = r - l, r--;
      }
    }
  }

  return z_table;
}

vector<int> z_table(string &txt, string &pat) {
  string concat = pat + "|" + txt;
  int l = int(concat.size());

  vector<int> Z = get_z_table(concat);

  int d = int(pat.size()) + 1;
  vector<int> z_table(int(txt.size()));
  for (int i = d; i < l; ++i)
    z_table[i - d] = Z[i];

  z_table.front() = 0;
  return z_table;
}
```

➢ KMP

```
vector<int> buildPI(string &pat) {
  int m = pat.size();
  vector<int> pi(m);
  for (int i = 1, c = 0; i < m; i++) {
    while (c > 0 && pat[c] != pat[i])
      c = pi[c - 1];

    if (pat[c] == pat[i]) pi[i] = ++c;
    else pi[i] = c;
  }
  return pi;
}

void KMP(string &s, string &pat, vector<int> &id) {
  int n = s.size(), m = pat.size();
  vector<int> pi = buildPI(pat);

  for (int i = 0, c = 0; i < n; i++) {
    while (c > 0 && pat[c] != s[i])
      c = pi[c - 1];

    if (pat[c] == s[i])
      c++;

    if (c == m) {
      id.push_back(i - m + 1);
      c = pi[c - 1];
    }
  }
}
```

➢ Arithmetic Progression

```
pair<vector<int>, vector<int>> build(vector<int> &v) {
  int n = v.size();
  vector<int> suff(n + 2), s(n + 2);
  for (int i = n - 1; i >= 0; --i) {
    suff[i] = suff[i + 1] + v[i];
    s[i] = s[i + 1] + suff[i];
  }

  return {suff, s};
}

int get(int l, int r, vector<int> &suff, vector<int> &s) {
  int res = s[l] - s[r + 1];
  res -= (r - l + 1) * suff[r + 1];
  return res;
}
```

➢ Merge Segments

```
void merge(vector<pair<int, int>> &vc) {
  sort(vc.begin(), vc.end());
  vector<int> start, end;
  int st = vc[0].first, en = vc[0].second;
  for (int i = 1; i < vc.size(); i++) {
    if (vc[i].first <= en) en = max(en, vc[i].second);
    else {
      start.push_back(st);
      end.push_back(en);
      st = vc[i].first;
      en = vc[i].second;
    }
  }
  start.push_back(st);
  end.push_back(en);
}
```

➢ Number of mod in range

```
// count number of y such that y%n = x  (0 -> l)

int f(int l, int n, int x) {
  int cnt = (l / n) + (l % n >= x);
  return cnt;
}
```

➢ Kadane on Matrix

```
int kadane(vector<int> &v) {
  int s = 0, mx = -oo;
  for (int &i: v) {
    s += i;
    mx = max(s, mx);
    s = max(0ll, s);
  }
  return mx;
}

int maxSubmatrixSum(vector<vector<int>> &A) {
  int r = A.size();
  int c = A[0].size();
  int **pre = new int *[r];

  for (int i = 0; i < r; i++) {
    pre[i] = new int;
    for (int j = 0; j < c; j++)
      pre[i][j] = 0;
  }

  for (int i = 0; i < r; i++) {
    for (int j = 0; j < c; j++) {
      pre[i][j] = A[i][j];
      if (i)
        pre[i][j] += pre[i][j - 1];
    }
  }

  int maxSum = -oo;
  for (int i = 0; i < c; i++) {
    for (int j = i; j < c; j++) {
      vector<int> v;
      for (int k = 0; k < r; k++) {
        int el = pre[k][j];
        if (i) el -= pre[k][i - 1];
        v.push_back(el);
      }
      maxSum = max(maxSum, kadane(v));
    }
  }

  return maxSum;
}
```

➢ 2D Prefix Sum

```
// g is one indexed!!!
vector<vector<int>> g(n+1, vector<int>(m+1));

for(int i = 1; i <= n; i++)
  for(int j = 1; j <= m; j++)
    g[i][j] = g[i][j-1] + mat[i-1][j-1];

for(int j = 1; j <= m; j++)
  for(int i = 1; i <= n; i++)
    g[i][j] += g[i-1][j];

while(q--) {
  int a, b, c, d; cin >> a >> b >> c >> d;
  cout << g[c][d] + g[a - 1][b - 1] - g[a - 1][d] - g[c][b - 1]
<< endl;
}
```

➢ Montonic Stack

```
vector<int> nextGreater(vector<int> &v) {
  int n = v.size();
  vector<int> res(n, n);
  stack<int> st;
  for (int i = 0; i < n; ++i) {
    if (st.empty() || v[i] <= v[st.top()]) st.push(i);
    else {
      res[st.top()] = i;
      st.pop(), --i;
    }
  }
  return res;
}
vector<int> prevGreater(vector<int> &v) {
  int n = v.size();
  vector<int> res(n, -1);
  stack<int> st;
  for (int i = n - 1; i >= 0; --i) {
    // you may need to remove the equal
    if (st.empty() || v[i] <= v[st.top()]) st.push(i);
    else {
      res[st.top()] = i;
      st.pop(), ++i;
    }
  }
  return res;
```

➢ STL Compare

```
struct compare {
  // right is the top -> pq
  // left is the begin -> set, map
  // return false if equal
  bool operator()(# a, # b) const {

  }
};
```

➢ Minimum Swaps

```
int cost(vector<int>& from, vector<int>& to) {
  int n = int(from.size());

  vector<int, int> mp;
  ordered_set<int> st;
  for (int i = 0; i < n; ++i) {
    st.insert(i);
    mp[from[i]] = i;
  }

  int ret = 0;
  for (int i = 0; i < n; ++i) {
    ret += st.order_of_key(mp[to[i]]);
    st.erase(mp[to[i]]);
  }

  return ret;
}
```

➢ Mod Operations & Comb

```
// make sure a & b < mod
int add(int a, int b, int mod) {
  ll res = (ll) a + b;
  if (res >= mod) res -= mod;
  if (res < 0) res += mod;
  return res;
}

int mul(int a, int b, int mod) {
  return (1LL * a * b) % mod;
}

int power(int a, int b, int mod) {
  int ret = 1;
  while (b) {
    if (b & 1) ret = 1ll * ret * a % mod;
    a = 1ll * a * a % mod, b >>= 1;
  }
  return ret;
}

int inverse(int b, int mod) {
  return power(b, mod - 2, mod);
}

struct Comb {
  vector<int> fact; // pre process fact inv too :)
  Comb(int n) {
    fact.assign(n + 5, 1);
    for (int i = 1; i <= n; ++i) fact[i] = mul(i, fact[i - 1]);
  }

  int nPr(int n, int r) {
    return n < r ? 0 : mul(fact[n], inverse(fact[n - r]));
  }

  int nCr(int n, int r) {
    return mul(nPr(n, r), inverse(fact[r]));
  }
};
```

➢ Linear Sieve

```
// pr is all the primes, low[x] is the lowest prime of x
vector<int> pr, low;
void Sieve(int n) {
  low.assign(n + 1, 0);
  for (int i = 2; i <= n; ++i) {
    if (!low[i]) {
      low[i] = i;
      pr.push_back(i);
    }

    for (int &j: pr) {
      if (j > low[i] || i * j > n) break;
      low[j * i] = j;
    }
  }
}
```

➢ Segmented Sieve

```
vector<bool> segmentedSieve(int L, int R) {
  // generate all primes up to sqrt(R)
  int lim = sqrtl(R);
  vector<bool> mark(lim + 1, false);
  vector<int> primes;
  for (int i = 2; i <= lim; ++i) {
    if (!mark[i]) {
      primes.emplace_back(i);
      for (int j = i * i; j <= lim; j += i)
        mark[j] = true;
    }
  }
  vector<bool> isPrime(R - L + 1, true);
  for (int i: primes)
    for (int j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
      isPrime[j - L] = false;
  if (L == 1)
    isPrime[0] = false;
  return isPrime;
}
```

➢ Count Divisors up to 1e18

```
N = input()
primes = array containing primes till 10^6
ans = 1
for all p in primes :
            if p*p*p > N:
                  break
            count = 1
            while N divisible by p:
                  N = N/p
                  count = count + 1
            ans = ans * count
if N is prime:
            ans = ans * 2
else if N is square of a prime:
            ans = ans * 3
else if N != 1:
            ans = ans * 4
```

➢ Int128

```
__int128 read() {
  __int128 x = 0, f = 1;
  char ch = getchar();
  while (ch < '0' || ch > '9') {
    if (ch == '-') f = -1;
    ch = getchar();
  }
  while (ch >= '0' && ch <= '9') {
    x = x * 10 + ch - '0';
    ch = getchar();
  }
  return x * f;
}

void print(__int128 x) {
  if (x < 0) {
    putchar('-');
    x = -x;
  }
  if (x > 9) print(x / 10);
  putchar(x % 10 + '0');
}
```

➢ Prime Tester

```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(1.L / M * a * b);
  return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = modmul(b, b, mod), e /= 2)
    if (e & 1) ans = modmul(ans, b, mod);
  return ans;
}

bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ull A[] = {2, 325, 9375, 28178, 450775, 9780504,
1795265022},
      s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : A) {
    ull p = modpow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = modmul(p, p, n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

➢ Sieve up to 1e9

```
vector<int> sieve(const int N, const int Q = 17, const
int L = 1 << 15) {
  static const int rs[] = {1, 7, 11, 13, 17, 19, 23, 29};
  struct P {
    P(int p) : p(p) {}
    int p;
    int pos[8];
  };
  auto approx_prime_count = [](const int N) -> int {
    return N > 60184 ? N / (log(N) - 1.1)
                : max(1., N / (log(N) - 1.11)) + 1;
  };
  const int v = sqrt(N), vv = sqrt(v);
  vector<bool> isp(v + 1, true);
  for (int i = 2; i <= vv; ++i)
    if (isp[i]) {
      for (int j = i * i; j <= v; j += i) isp[j] = false;
    }
  const int rsize = approx_prime_count(N + 30);
  vector<int> primes = {2, 3, 5};
  int psize = 3;
  primes.resize(rsize);
  vector<P> sprimes;
  size_t pbeg = 0;
  int prod = 1;
  for (int p = 7; p <= v; ++p) {
    if (!isp[p]) continue;
    if (p <= Q) prod *= p, ++pbeg, primes[psize++] = p;
    auto pp = P(p);
    for (int t = 0; t < 8; ++t) {
      int j = (p <= Q) ? p : p * p;
      while (j % 30 != rs[t]) j += p << 1;
      pp.pos[t] = j / 30;
    }
    sprimes.push_back(pp);
  }
  vector<unsigned char> pre(prod, 0xFF);
  for (size_t pi = 0; pi < pbeg; ++pi) {
    auto pp = sprimes[pi];
    const int p = pp.p;
    for (int t = 0; t < 8; ++t) {
      const unsigned char m = ~(1 << t);
      for (int i = pp.pos[t]; i < prod; i += p) pre[i] &= m;
    }
  }
  const int block_size = (L + prod - 1) / prod * prod;
  vector<unsigned char> block(block_size);
  unsigned char *pblock = block.data();
  const int M = (N + 29) / 30;
  for (int beg = 0; beg < M; beg += block_size, pblock -
= block_size) {
    int end = min(M, beg + block_size);
    for (int i = beg; i < end; i += prod)
      copy(pre.begin(), pre.end(), pblock + i);
    if (beg == 0) pblock[0] &= 0xFE;
    for (size_t pi = pbeg; pi < sprimes.size(); ++pi) {
      auto &pp = sprimes[pi];
      const int p = pp.p;
      for (int t = 0; t < 8; ++t) {
        int i = pp.pos[t];
        const unsigned char m = ~(1 << t);
        for (; i < end; i += p) pblock[i] &= m;
        pp.pos[t] = i;
      }
    }
    for (int i = beg; i < end; ++i)
      for (int m = pblock[i]; m > 0; m &= m - 1)
        primes[psize++] = i * 30 + rs[__builtin_ctz(m)];
  }
  while (psize > 0 && primes[psize - 1] > N) --psize;
  primes.resize(psize);
  return primes;
}
```

➢ Math Formulas & Theoroms

9. $\sum_{i=1}^{n} \frac{i}{2^i} = 2^{-n}(-n + 2^{n+1} - 2)$

Sum

$$\sum_{i=1}^{n} i\, b^i = \frac{b\,(n\, b^{n+1} - (n+1)\, b^n + 1)}{(b-1)^2}$$

```
summation from l to r with step (sum of numbers divisible by step in range)

ll calc(int l, int r, int step) {
    --l;
    ll sum = 1ll * step * (r / step) * ((r / step) + 1) / 2LL;
    sum -= 1ll * step * (l / step) * ((l / step) + 1) / 2LL;
    return sum;
}
```

For two coprime positive integers m and n, that largest number that cannot be written as am + bn (cannot be made by adding up m and n) will be m*n — m — n.

Cayley's formula is the number of spanning trees of N nodes which is equal to N^{N-2}

Number of spanning trees in complete bipartite graph $G_{X,Y}$ is : $X^{Y-1} \times Y^{X-1}$

Such that

X : is the number of nodes in the first set.

Y : is the number of nodes in the second set.

```
int SumOfOddCubes(int n) {return n * n * (2 * n * n - 1);}

int SumOfOddSquares(int n) {return n * (2 * n + 1) * (2 * n - 1) / 3;}

int SumOfEvenCubes(int n) {return 2 * n * n * (n + 1) * (n + 1);}

int SumOfEvenSquares(int n) {return 2 * n * (n + 1) * (2 * n + 1) / 3;}
```

$$\sum_{k=1}^{n} k = \frac{1}{2}\left(n^2 + n\right)$$

$$\sum_{k=1}^{n} k^2 = \frac{1}{6}\left(2n^3 + 3n^2 + n\right)$$

$$\sum_{k=1}^{n} k^3 = \frac{1}{4}\left(n^4 + 2n^3 + n^2\right)$$

$$\sum_{k=1}^{n} k^4 = \frac{1}{30}\left(6n^5 + 15n^4 + 10n^3 - n\right)$$

$$\sum_{k=1}^{n} k^5 = \frac{1}{12}\left(2n^6 + 6n^5 + 5n^4 - n^2\right)$$

$$\sum_{k=1}^{n} k^6 = \frac{1}{42}\left(6n^7 + 21n^6 + 21n^5 - 7n^3 + n\right)$$

$$\sum_{k=1}^{n} k^7 = \frac{1}{24}\left(3n^8 + 12n^7 + 14n^6 - 7n^4 + 2n^2\right)$$

$$\sum_{k=1}^{n} k^8 = \frac{1}{90}\left(10n^9 + 45n^8 + 60n^7 - 42n^5 + 20n^3 - 3n\right)$$

$$\sum_{k=1}^{n} k^9 = \frac{1}{20}\left(2n^{10} + 10n^9 + 15n^8 - 14n^6 + 10n^4 - 3n^2\right)$$

$$\sum_{k=1}^{n} k^{10} = \frac{1}{66}\left(6n^{11} + 33n^{10} + 55n^9 - 66n^7 + 66n^5 - 33n^3 + 5n\right).$$

- The number of factors of n is : $\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1),$
- Sum of factors of n : $\sigma(n) = \prod_{i=1}^{k} (1 + p_i + \ldots + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1},$
- The product of factors : $\mu(n) = n^{\tau(n)/2}$

➢ Theorem (Fermat).

Every prime of the form 4k +1 is the sum of two squares. A positive integer n is the sum of two squares if and only if all prime factors of the form 4k - 1 have an even exponent in the prime- factorization of n.

Legendre's three-square theorem

Let n be a natural number. $n = x^2 + y^2 + z^2$ is solvable in non-negative number

$\iff$ n is not of the following form: $4^a(8b + 7)$

Lagrange's four-square theorem

Every natural number can be represented as a sum of four non-negative integer squares.

If $n = 4^a(8b + 7)$, applying Lagrange's four-square theorem return 4.

Otherwise answer could be 1, 2 or 3.

Counting

Combinatorics

- nPr

$$\frac{!n}{!(n-r)}$$

- nCr

$$\frac{nPr}{!r} = \frac{!n}{!r*!(n-r)}$$

Recurrance relation

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

- **Stars and Bars**

  the number of ways to distribute n balls in k boxes

  suppose you have k - 1 bars that and you want to place them between the balls

$$\binom{n+k-1}{k-1}$$

- Lucas Theorem

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p},$$

  - `mi` is the coefficient of the number `m` after converting it to base p (the digits it self)
  - `ni` is the coefficient of the number `n` after converting it to base p (the digits it self)

▼ Hockey Stick Identity

$$\sum_{k=r}^{n} \binom{k}{r} = \binom{n+1}{r+1}$$

▼ Derangements

the number of ways to shuffle a set consisting of `n` elements

no element stays in his initial position

$$D(n) = (n-1) * (D(n-1) + D(n-2))$$

Counting Tricks

1. Dearrangements :
   1st : NC0 * N! - NC1 * (N-1)! + NC2 * (N-2)! - NC3 * (N-3)! + ... +NCN * 0!
   2st : Dn = (n - 1)(Dn-1 + Dn-2)
   to set k fixed points --> (N)C(K) * D(N-K)

2. NcR = N-1cR + N-1cR-1
3. stars and bars:
   N balls, K baskets : N+K-1 c K-1
   if no basket can have 0 balls : N-1 c K-1
   we can multiply this by N! to get all permutations of it
4. The last digit in the a^b is repeated every 4 times
5. To choose things in increasing order -> use combinations
6. Lattice grid -> number of ways to go from (x1, y1) to (x2, y2)
   a. (x2+y2) – (x1+y1) C (x2 – x1)
7. If we can repeat choosing the element we must use stars and bars not combination (bars represent the switch between element and the other).
8. Number of multisets **[with size(k)]** that have numbers from **0** to **n (n+1 numbers)**
   a. (n + k) C n
9. $\sum_{i=1}^{n} \frac{i}{2^i} = 2^{-n}(-n + 2^{n+1} - 2)$
10. Arithmetic progression:

$$S_n = \frac{n}{2}[2a + (n-1)d].$$

This formula can be simplified as:

$$\begin{aligned} S_n &= \frac{n}{2}[a + a + (n-1)d]. \\ &= \frac{n}{2}(a + a_n). \\ &= \frac{n}{2}(\text{initial term} + \text{last term}). \end{aligned}$$

For a geometric Progression a, ar, ar², ar³ ...

- nth Term,

$$a_n = r\, a_{n-1}$$

- Sum of n terms

$$S_n = \begin{cases} \frac{a(r^n - 1)}{r - 1} & \text{,when } r \neq 1 \\ na & \text{,when } r \neq 1 \end{cases}$$

- Sum of infinite terms

$$S_n = \begin{cases} \frac{a}{1 - r} & \text{,When } |r| < 1. \\ \text{diverges ,When } |r| \geq 1. \end{cases}$$

➢ FFT

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
  int n = a.size();

  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1)
      j ^= bit;
    j ^= bit;

    if (i < j)
      swap(a[i], a[j]);
  }

  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI / len * (invert ? -1 : 1);
    cd wlen(cos(ang), sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
        cd u = a[i+j], v = a[i+j+len/2] * w;
        a[i+j] = u + v;
        a[i+j+len/2] = u - v;
        w *= wlen;
      }
    }
  }

  if (invert) {
    for (cd & x : a)
      x /= n;
  }
}

vector<int> multiply(vector<int> const& a,
vector<int> const& b) {
  vector<cd> fa(a.begin(), a.end()), fb(b.begin(),
b.end());
  int n = 1;
  while (n < a.size() + b.size())
    n <<= 1;
  fa.resize(n);
  fb.resize(n);

  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++)
    fa[i] *= fb[i];
  fft(fa, true);

  vector<int> result(n);
  for (int i = 0; i < n; i++)
    result[i] = round(fa[i].real());
  return result;
}
```

➢ FFTMOD

```
#define rep(aa, bb, cc) for(int aa = bb; aa < cc;aa++)
#define sz(a) (int)a.size()
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
  }
  vi rev(n);
  rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
        // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-
rolled)  /// include-line
        auto x = (double *)&rt[j+k], y = (double
*)&a[i+j+k];      /// exclude-line
        C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
/// exclude-line
        a[i + j + k] = a[i + j] - z;
        a[i + j] += z;
      }
}

template<int M> vi convMod(const vi &a, const vi &b)
{
  if (a.empty() || b.empty()) return {};
  vi res(sz(a) + sz(b) - 1);
  int B=32-__builtin_clz(sz(res)), n=1<<B,
cut=int(sqrt(M));
```

```
  vector<C> L(n), R(n), outs(n), outl(n);
  rep(i,0,sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  rep(i,0,sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  rep(i,0,n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  rep(i,0,sz(res)) {
    ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])+.5);
    ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
    res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
  }
  return res;
}
```

➢ NTT

```
const int mod = 998244353;
const int root = 31;
const int root_1 = 128805723;
const int root_pw = 1 << 23;

// make the size of (a) power of 2
void fft(vector<int> &a, bool invert) {
  int n = a.size();

  // reversal bit sort
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1)
      j ^= bit;
    j ^= bit;

    if (i < j)
      swap(a[i], a[j]);
  }

  for (int len = 2; len <= n; len <<= 1) {
    int wlen = invert ? root_1 : root;
    for (int i = len; i < root_pw; i <<= 1)
      wlen = 1ll * wlen * wlen % mod;
    for (int i = 0; i < n; i += len) {
      int w = 1;
      for (int j = 0; j < len / 2; j++) {
        int u = a[i + j], v = 1ll * a[i + j + len / 2] * w %
mod;
        a[i + j] = u + v;
        if (a[i + j] >= mod) a[i + j] -= mod;
        a[i + j + len / 2] = u - v;
        if (a[i + j + len / 2] < 0) a[i + j + len / 2] += mod;
        w = 1ll * w * wlen % mod;
      }
    }
  }

  if (invert) {
    int n_1 = power(n, mod - 2);
    for (int &x: a)
      x = 1ll * x * n_1 % mod;
  }
}


vector<int> multiply(vector<int> const &a,
vector<int> const &b) {
  vector<int> fa = a, fb = b;
  int n = 1;
  while (n < a.size() + b.size())
    n <<= 1;
  fa.resize(n), fb.resize(n);

  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++)
    fa[i] = 1ll * fa[i] * fb[i] % mod;
  fft(fa, true);

  fa.resize(a.size() + b.size());
  return fa;
}

vector<int> power(vector<int> &a, int b) {
  vector<int> res{1};
  while (b) {
    if (b & 1) res = multiply(res, a);
    a = multiply(a, a), b >>= 1;
  }
  return res;
}
```

➢ phi function

```
// phi[p] = p - 1
// phi[p^k] = p^k - p^(k - 1)
// phi[a * b] = phi[a] * phi[b]
// sum:[d|n] phi[d] = n
// a^(phi[m]) = 1 % m
// a^(n) = a^(n % phi[m]) % m

long long phi(long long n) {
  long long ans = n;
  for (int p = 2; 1LL * p * p <= n; p++) {
    if (n % p == 0) {
      while (n % p == 0) {
        n /= p;
      }
      ans -= ans / p;
    }
  }
  if (n > 1) {
    ans -= ans / n;
  }
  return ans;
}


const int N = #;

int phi[N];

void calc_phi() {
  for (int i = 1; i < N; i++) {
    phi[i] = i;
  }
  for (int i = 2; i < N; i++) {
    if (phi[i] == i) {
      for (int j = i; j < N; j += i) {
        phi[j] -= phi[j] / i;
      }
    }
  }
}
```

➢ extended eculidian

```
// x * a + y * b = g
// x = x0 + b / g * t
// y = y0 - a / g * t

int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);
  }
  return a1;
}

// extended (a, m, x, y) -> inverse of a
if (g != 1) {
cout << "No solution!";
} else {
x = (x % m + m) % m;
cout << x << endl;
}

vector<int> invs(vector<int> a, int mod) {
  int n = int(a.size());
  vector<int> ret(n);
  int v = 1;
  for (int i = 0; i != n; ++i) {
    ret[i] = v;
    v = 1LL * v * a[i] % mod;
  }
  auto [x, y] = extended_gcd(v, mod);
  x = (x % mod + mod) % mod;
  for (int i = n - 1; i >= 0; --i) {
    ret[i] = 1LL * x * ret[i] % mod;
    x = 1LL * x * a[i] % mod;
  }
  return ret;
}
```

➢ Mobius

```
mobius[1] = -1;
for (int i = 1; i < VALMAX; i++) {
  if (mobius[i]) {
    mobius[i] = -mobius[i];
    for (int j = 2 * i; j < VALMAX; j += i) {
      mobius[j] += mobius[i];
    }
  }
}
```

➢ Pascal

```
struct pascal_triangel {
  vector<vector<int>> nCr;

  pascal_triangel(int n) {
    nCr = vector<vector<int>>(n + 1, vector<int>(n +
1));

    nCr[0][0] = 1;
    for (int i = 1; i <= n; i++) {
      nCr[i][0] = 1;
    }
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= i; j++) {
        // mod overflow
        nCr[i][j] = nCr[i - 1][j] + nCr[i - 1][j - 1];
      }
    }
  }
};

pascal_triangel g(#);
```