# UniTrento Musketrees Notebook



# Graphs

## Tarjan SCC

```
typedef vector<int> vi;
typedef vector<vi> vvi;
#define MAX 100005
int n, m,foundat=1;
vvi graph,scc;
vi disc,low; // init disc to -1
bool onstack[MAX]; //init to 0

void tarjan(int u){
   static stack<int> st;
   disc[u]=low[u]=foundat++;
   st.push(u);
   onstack[u]=true;
   for(auto i:graph[u]){
       if(disc[i]==-1){
           tarjan(i);
           low[u]=min(low[u],low[i]);
       }
       else if(onstack[i])
           low[u]=min(low[u],disc[i]);
   }
   if(disc[u]==low[u]){
       vi scctem;
       while(1){
           int v=st.top();
           st.pop();onstack[v]=false;
           scctem.push_back(v);
           if(u==v)
               break;
       }
       scc.push_back(scctem);
   }
}
int main() {
   // n= vertices of graph (1 based)
   graph.clear();graph.resize(n+1);
   disc.clear();disc.resize(n+1,-1);
   low.clear();low.resize(n+1);
   // input graph here
   for(int i = 0; i < n; i++)
       if(disc[i+1]==-1)
           tarjan(i+1);
   for(auto i:scc)
       for(auto j:i)
           // iterate over the vertices in i
}
```

## Articulation points and bridges

Articulation points

```
ii dfs_points(vector<vector<int>> &graph,
vector<bool> &visited, vector<int> &id, vector<int>
&low, int root, int current, int parent, int
curr_id, vector<bool> &art_points, ii &result){
        visited[current] = true;
        if(parent == root)
            result.second++;
```

```
        low[current] = id[current] = ++curr_id;
        result.first = curr_id;
        for(int edge : graph[current]){
        if(edge == parent) continue;
        if(!visited[edge]){
                result = dfs_points(graph, visited,
id, low, root, edge, current, result.first,
art_points, result);
                low[current] = min(low[current],
low[edge]);
                if(id[current] <= low[edge])
                art_points[current] = true;
        }else
                low[current] = min(low[current],
id[edge]);
        }
        return result;
}

vector<bool> articulation_points(vector<vector<int>>
&graph){
        vector<bool> art_points(graph.size(), false);
        vector<int> id(graph.size(), 0);
        vector<int> low(graph.size(), 0);
        vector<bool> visited(graph.size(), false);

        for(int i=0; i<graph.size(); i++)
        if(!visited[i]){
                ii result(0,0);
                result = dfs_points(graph, visited,
id, low, i, i, -1, result.first, art_points,
result);
                art_points[i] = (result.second > 1);
        }

        return art_points;
}
```

Bridges

```
int dfs_bridges(vector<vector<int>> &graph,
vector<bool> &visited, vector<int> &id, vector<int>
&low, int node, int parent, int curr_id, vector<ii>
&bridges){

        visited[node] = true;
        low[node] = id[node] = ++curr_id;

        for(int adj : graph[node]){
            if(adj == parent) continue;
            if(!visited[adj]){
                curr_id = dfs_bridges(graph, visited,
id, low, adj, node, curr_id, bridges);
                low[node] = min(low[node], low[adj]);
                if(id[node] < low[adj])
                    bridges.push_back(make_pair(node,
adj));
            }else{
                low[node] = min(low[node], id[adj]);
            }
        }
        return curr_id;
}

vector<ii> find_bridges(vector<vector<int>> &graph){

        vector<ii> bridges;
        vector<int> id(graph.size(), 0);
        vector<int> low(graph.size(), 0);
        vector<bool> visited(graph.size(), false);

        int curr_id = -1;
        for(int i=0; i<graph.size(); i++)
        if(!visited[i])
                curr_id = dfs_bridges(graph, visited,
id, low, i, -1, curr_id, bridges);

        return bridges;
}
```

## Union Find Set

```
struct UnionFind {
   vi p, rank, setSize;
   int numSets;
   UnionFind(int N) {
       setSize.assign(N, 1);
       numSets = N; rank.assign(N, 0);
       p.assign(N, 0);
       for (int i = 0; i < N; i++) p[i] = i;
   }
   int findSet(int i) {
       return (p[i] == i) ? i : (p[i] =
findSet(p[i]));
   }
   bool isSameSet(int i, int j) { return findSet(i)
== findSet(j); }
   void unionSet(int i, int j) {
       if (!isSameSet(i, j)) {
           numSets--;
           int x = findSet(i), y = findSet(j);
           // rank is used to keep the tree short
           if (rank[x] > rank[y]) {
               p[y] = x;
               setSize[x] += setSize[y];
           } else {
               p[x] = y; setSize[y] += setSize[x];
               if (rank[x] == rank[y]) rank[y]++;
           }
       }
   }
   int numDisjointSets() { return numSets; }
   int sizeOfSet(int i) {
       return setSize[findSet(i)];
   }
};
```

## Minimum spanning tree (Kruskal + Prim)

```
vector<vii> AdjList;
vi taken;
```

```
priority_queue<ii, vector<ii>, greater<ii>> pq;
void process(int vtx) {
   taken[vtx] = 1;
   for (int j = 0; j < AdjList[vtx].size(); j++) {
       ii v = AdjList[vtx][j];
       if (!taken[v.first])
           pq.push(ii(v.second, v.first));
   }
}

int main() {
   int V, E, u, v, w;
   scanf("%d %d", &V, &E);
   AdjList.assign(V, vii());
   vector< pair<int, ii> > EdgeList;
   for (int i = 0; i < E; i++) {
       scanf("%d %d %d", &u, &v, &w);
       EdgeList.push_back(make_pair(w, ii(u, v)));
       AdjList[u].push_back(ii(v, w));
       AdjList[v].push_back(ii(u, w));
   }

   // Kruskal
   sort(EdgeList.begin(), EdgeList.end());
   int mst_cost = 0;
   UnionFind UF(V);
   for (int i = 0; i < E; i++) {
       pair<int, ii> front = EdgeList[i];
       if (!UF.isSameSet(front.second.first,
front.second.second)) {
           mst_cost += front.first;
           UF.unionSet(front.second.first,
front.second.second);
       }
   }
   printf("MST cost = %d (Kruskal's)\n", mst_cost);
   // Prim
   taken.assign(V, 0);
   process(0);
   mst_cost = 0;
   while (!pq.empty()) {
       ii front = pq.top(); pq.pop();
       u = front.second, w = front.first;
       if (!taken[u])
           mst_cost += w, process(u);
   }
   printf("MST cost = %d (Prim's)\n", mst_cost);
   return 0;
}
```

## Dijkstra

```
#define INF 1000000000
vector<vii> AdjList;
void dijkstra(int s, vi &dist){
   dist.assign(V, INF);
   dist[s] = 0;
   priority_queue<ii, vector<ii>, greater<ii> > pq;
   pq.push(ii(0, s));
   while (!pq.empty()) {
       ii front = pq.top(); pq.pop();
       int d = front.first, u = front.second;
       if (d > dist[u]) continue;
       for (int j = 0; j<AdjList[u].size(); j++) {
           ii v = AdjList[u][j];
           if (dist[u]+v.second < dist[v.first]) {
               dist[v.first] = dist[u] + v.second;
               pq.push(ii(dist[v.first], v.first));
           }
       }
   }
}
```

## Bellman Ford

```
void BellmanFord(int s, vi &dist){
       dist.assign(V, INF);
       dist[s] = 0;
       for (int i = 0; i < V - 1; i++)
       for (int u = 0; u < V; u++)
           for(int j=0;j<AdjList[u].size();j++){
               ii v = AdjList[u][j];
               dist[v.first] = min(dist[v.first],
dist[u]+v.second);
            }
       bool hasNegativeCycle = false;
       for (int u = 0; u < V; u++)
           for (int j=0; j<AdjList[u].size();j++){
               ii v = AdjList[u][j];
               if (dist[v.first]>dist[u] + v.second)
                   hasNegativeCycle = true;
       }
}
```

## Floyd Warshall

```
int V, E, u, v, w, A[200][200];
int main() {
   scanf("%d %d", &V, &E);
   for (int i = 0; i < V; i++) {
       for (int j = 0; j < V; j++)
           A[i][j] = INF;
       A[i][i] = 0;
   }
   for (int i = 0; i < E; i++) {
       scanf("%d %d %d", &u, &v, &w);
       A[u][v] = w;
   }
   for (int k = 0; k < V; k++)
       for (int i = 0; i < V; i++)
           for (int j = 0; j < V; j++)
               A[i][j]=min(A[i][j],A[i][k] +
A[k][j]);
   // negative cycle if A[i][i] < 0
   return 0;
}
```

## Edmonds - Karp

```
struct NetFlow{
   #define ii pair<int, int>
   vector<unordered_map<int, ii>> adj;
   vector<int> parent;
   int N;
   NetFlow(int n): N(n), adj(n), parent(n){}
```

```
    void incCapacity(int a, int b, int c){
        adj[a][b].first+=c;
    }
    void addFlow(int a, int b, int flow){
        adj[a][b].first  -=flow; //capacità attuale
        adj[a][b].second +=flow; //capacità
residua(positivo)/flusso
        adj[b][a].first  +=flow;
        adj[b][a].second -=flow;
    }
    int bfsTparent(int start, int end){
        vector<bool> vis(N, false);
        queue<ii> q({ii(start, 1<<30)});//INT_MAX
        vis[start]=true;
        for(ii u=q.front(); !q.empty(); q.pop(),
u=q.front())
            for(pair<int, ii> v: adj[u.first])
                if(!vis[v.first] &&
v.second.first>0){
                    vis[v.first]=true;
                    parent[v.first]=u.first;
                    if(v.first==end)
                        return min(u.second,
v.second.first);
                    q.push(ii(v.first, min(u.second,
v.second.first)));
                }
        return 0;
    }
    int open(int start, int end){//ritorna l'aumento
di flusso
        int augment=0;
        if(start==end) return 0;
        for(int f; f=bfsTparent(start, end);
augment+=f)
            for(int v = end; v != start; v =
parent[v])
                addFlow(parent[v], v,-f);
        return augment;
    }
};
```

## Dinic

```
// Max Flow : Dinic's algorithm
// runs in
//      * O(V^2 E) time in general graph
//      * O(min(V^(2/3), E^(1/2)) * E) time in unit
capacity network
//      * O(V^(1/2) * E) time in unit capacity simple
network

struct MaxFlow{
    struct edge{int next, inv, cap, orig;};
    int n;
    vector<vector<edge>> g;
    vector<int> l, cur;
    const int INF = 987654321;
    void add_edge(int u, int v, int w, int wr = 0){
        edge forward = {v, (int)g[v].size(), w, w};
        edge backward = {u, (int)g[u].size(), wr,
wr};
        g[u].push_back(forward);
        g[v].push_back(backward);
    }
    MaxFlow(int _n) : n(_n) {
        g.resize(n);
        l.resize(n);
        cur.resize(n);
    }
    bool construct_level_graph(int src, int dst){
        fill(l.begin(), l.end(), -1);
        fill(cur.begin(), cur.end(), 0);
        l[src] = 0;
        queue<int> q;
        q.push(src);
        while(!q.empty()){
            int here = q.front(); q.pop();
            for(auto& e : g[here])
                if(e.cap > 0 && l[e.next] == -1){
                    l[e.next] = l[here] + 1;
                    q.push(e.next);
                }
        }
        return l[dst] != -1;
    }
    int dfs(int here, int dst, int flow) {
        if(here == dst) return flow;
        for(int& i=cur[here];i<g[here].size();i++){
            auto& e = g[here][i];
            if(e.cap > 0 && l[here] + 1 ==
l[e.next]){
                int
f=dfs(e.next,dst,min(flow,e.cap));
                if(f > 0){
                    e.cap -= f;
                    g[e.next][e.inv].cap += f;
                    return f;
                }
            }
        }
        return 0;
    }
    int solve(int src, int dst){
        int total_flow = 0;
        while(construct_level_graph(src, dst)){
            while(true){
                int f = dfs(src, dst, INF);
                if(f == 0) break;
                total_flow += f;
            }
        }
        return total_flow;
    }
};

int main(){
    int n;
    MaxFlow mf(2*n+2);
    int src = 2*n, dst = 2*n+1;
    while(scanf("%d", &n) > 0){
        for(int i=0;i<n;i++){
```

```
            int u, nr;
            scanf(" %d : ( %d )", &u, &nr);
            while(nr--){
                int v;
                scanf("%d", &v);
                mf.add_edge(u, v, 1);
            }
            mf.add_edge(src, i, 1);
            mf.add_edge(n+i, dst, 1);
        }
        printf("%d\n", mf.solve(src, dst));
    }
    return 0;
}
```

## Hopcroft Karp-Maximum Bipartite Matching

```
using namespace std;
#include <bits/stdc++.h>
struct HopcroftKarp {
    vector<int> vis, level, ml, mr;
    vector<vector<int>> g;
    int n, m;
    HopcroftKarp(int _n, int _m) {
        n = _n;
        m = _m;
        g.resize(n);
    }
    void addEdge(int u, int v) {
        g[u].push_back(v);
    }

    bool dfs(int u) {
        vis[u] = true;
        for (int x : g[u]) {
            int v = mr[x];
            if (v == -1 || (!vis[v] && level[u] <
level[v] && dfs(v))) {
                ml[u] = x;
                mr[x] = u;
                return true;
            }
        }
        return false;
    }

    int matching() {
        vis.resize(n, false);
        level.resize(n, 0);
        ml.resize(n, -1);
        mr.resize(m, -1);
        int d = 1, match;
        for (match = 0; d > 0; match += d) {
            queue<int> q;
            // bfs to find levels, start from free
nodes (not matched)
            for (int i = 0; i < n; ++i) {
                if (ml[i] == -1) {
                    level[i] = 0;
                    q.push(i);
                } else level[i] = -1;
            }
            while (!q.empty()) {
                int u = q.front(); q.pop();
                for (int x : g[u]) {
                    int v = mr[x];
                    if (v != -1 && level[v] < 0) {
                        level[v] = level[u] + 1;
                        q.push(v);
                    }
                }
            }
            vis.assign(n, false);
            d = 0;
            for (int i = 0; i < n; ++i)
                if (ml[i] == -1 && dfs(i))
                    ++d;
        }
        return match;
    }
};
```

## Hungarian - Min-cost Bipartite Matching

$O(n^3)$ implementation, it solves 1000x1000 problems in around 1 second.

cost[i][j] = cost for pairing left node i with right node j
Lmate[i] = index of right node that left node i pairs with
Rmate[j] = index of left node that right node j pairs The values in cost[i][j] may be positive or negative. To perform maximization, simply negate the cost[][] matrix.

```
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate,
VI &Rmate) {
    int n = int(cost.size());
    // construct dual feasible solution
    VD u(n), v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++)
            u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++)
            v[j] = min(v[j], cost[i][j] - u[i]);
    }
    // construct primal solution satisfying
complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if (fabs(cost[i][j]-u[i]-v[j]) < 1e-10) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
```

```
            }
        }
    }
    VD dist(n);
    VI dad(n);
    VI seen(n);
    // repeat until primal solution is feasible
    while (mated < n) {
        // find an unmatched left node
        int s = 0;
        while (Lmate[s] != -1) s++;
        // initialize Dijkstra
        fill(dad.begin(), dad.end(), -1);
        fill(seen.begin(), seen.end(), 0);
        for (int k = 0; k < n; k++)
            dist[k] = cost[s][k] - u[s] - v[k];
        int j = 0;
        while (true) {
            // find closest
            j = -1;
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                if (j==-1 || dist[k] < dist[j])
                    j = k;
            }
            seen[j] = 1;
            // termination condition
            if (Rmate[j] == -1) break;

            // relax neighbors
            const int i = Rmate[j];
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                const double new_dist = dist[j] +
cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;
                    dad[k] = j;
                }
            }
        }

        // update dual variables
        for (int k = 0; k < n; k++) {
            if (k == j || !seen[k]) continue;
            const int i = Rmate[k];
            v[k] += dist[k] - dist[j];
            u[i] -= dist[k] - dist[j];
        }
        u[s] += dist[j];

        // augment along path
        while (dad[j] >= 0) {
            const int d = dad[j];
            Rmate[j] = Rmate[d];
            Lmate[Rmate[j]] = j;
            j = d;
        }
        Rmate[j] = s;
        Lmate[s] = j;
        mated++;
    }
    double value = 0;
    for (int i = 0; i < n; i++)
        value += cost[i][Lmate[i]];

    return value;
}
```

## Min cost max flow

MCMF can be solved by replacing the O(E) BFS (to find the shortest - in terms of number of hops - augmenting path) in Edmonds Karp's algorithm into the O(V E) Bellman Ford's (to find the cheapest augmenting path). We need a shortest path algorithm that can handle negative edge weights as such negative edge weights may appear when we cancel a certain flow along a backward edge (as we have to subtract the cost taken by this augmenting path as canceling flow means that we do not want to use that edge). Time complexity ~ $O(V^2E^2)$.

# Dynamic Programming

## Knapsack

$$DP[i][c] = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \max(DP[i-1][c-w[i]] + p[i], DP[i-1][c]) & \text{otherwise} \end{cases}$$

## Matrix chain multiplication

$$DP[i][j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j}\{DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j\} & i < j \end{cases}$$

## Sliding window maximum

```
void printKMax(int arr[], int n, int k)  {
    deque<int> Qi(k);
    int i;
    for (i = 0; i < k; ++i) {
        while((!Qi.empty()) &&
arr[i]>=arr[Qi.back()])
            Qi.pop_back();
        Qi.push_back(i);
    }
    for (; i < n; ++i) {
        cout << arr[Qi.front()] << " ";
        while((!Qi.empty()) && Qi.front()<=i - k)
            Qi.pop_front();
        while((!Qi.empty()) &&
arr[i]>=arr[Qi.back()])
            Qi.pop_back();
        Qi.push_back(i);
    }
    cout << arr[Qi.front()];
}
```

# Strings

## Suffix Array

```
#include <bits/stdc++.h>
using namespace std;
#define MAX_N 1000000
char T[MAX_N];
int n;
```

```
int RA[MAX_N], tempRA[MAX_N];
int SA[MAX_N], tempSA[MAX_N];
int c[MAX_N];
int LCP[MAX_N], PLCP[MAX_N];
int Phi[MAX_N];

void countingSort(int k) {
   int i, sum, maxi = max(300, n);
   memset(c, 0, sizeof c);
   for (i = 0; i < n; i++)
       c[i + k < n ? RA[i + k] : 0]++;

   for (i = sum = 0; i < maxi; i++) {
       int t = c[i]; c[i] = sum; sum += t;
   }
   for (i = 0; i < n; i++)
       tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] =
SA[i];
   for (i = 0; i < n; i++)
       SA[i] = tempSA[i];
}

void constructSA() {
   int i, k, r;
   for (i = 0; i < n; i++) RA[i] = T[i];
   for (i = 0; i < n; i++) SA[i] = i;
   for (k = 1; k < n; k <<= 1) {
       countingSort(k);
       countingSort(0);
       tempRA[SA[0]] = r = 0;
       for (i = 1; i < n; i++)
           tempRA[SA[i]] =
           (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k]
== RA[SA[i-1]+k]) ? r : ++r;
       for (i = 0; i < n; i++)
           RA[i] = tempRA[i];

       if (RA[SA[n-1]] == n-1) break;
   }
}
void computeLCP() {
   int i, L;
   Phi[SA[0]] = -1;
   for (i = 1; i < n; i++)
       Phi[SA[i]] = SA[i-1];
   for (i = L = 0; i < n; i++) {
       if (Phi[i] == -1) { PLCP[i] = 0; continue; }
       while (T[i + L] == T[Phi[i] + L]) L++;
       PLCP[i] = L;
       L = max(L-1, 0);
   }
   for (i = 0; i < n; i++)
       LCP[i] = PLCP[SA[i]];
}
```

## Longest repeated substring (suffix)

```
int main(int argc, char const *argv[]) {
   string s; cin >> s; n = s.length();
   strcpy(T, s.c_str()); T[n] = '$';
   strcpy(T+n+1, s.c_str());
   int realN = n; n = 2*n + 1;
   constructSA();
   computeLCP();
   int cpindex, cplen = 0;
  // LRS check if not same string : same start
   for(int i = 1; i < n; i++)
       if(LCP[i] > cplen && ((SA[i-1] - realN - 1)
!= SA[i])){
           cplen = LCP[i];
           cpindex = i;
       }

   for(int i = SA[cpindex]; i < SA[cpindex] + cplen;
i++)
       printf("%c", T[i]);
   cout << endl;
   return 0;
}
```

## LCSubstring (suffix)

```
int main(int argc, char const *argv[]) {
   string s1, s2;
   int l1, l2;
   int rs;
   cin >> s1 >> s2;
   l1 = s1.length();
   l2 = s2.length();
   strcpy(T, s1.c_str());
   T[l1] = '$';
   strcpy(T+l1+1, s2.c_str());
   n = l1 + l2 + 1;
   constructSA();
   computeLCP();
   int len = 0;
   int maxindex = -1;
   for(int i = 1; i < n; i++){
       if(LCP[i] && !((SA[i-1] < l1 && SA[i] <l1) ||
(SA[i-1] < l2 && SA[i-1] > l1 && SA[i] <l2 && SA[i]
>l1))){
           if(LCP[i] > len){
               len = LCP[i];
               maxindex = i;
           }
       }
   }
   for(int i = SA[maxindex]; i < SA[maxindex] + len;
i++)
       printf("%c", T[i]);
   cout << endl;

   return 0;
}
```

## LCSubsequence

$$DP[i][j] = \begin{cases} 0 & i = 0 \textbf{ or } j = 0 \\ DP[i-1][j-1] + 1 & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i = u_j \\ \max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i \neq u_j \end{cases}$$

## Edit Distance

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & j = 0 \\ \min\{DP[i-1][j-1] + \delta, & \delta = \text{iif}(P[i] = T[j], 0, 1) \\ \quad DP[i-1][j] + 1, & \\ \quad DP[i][j-1] + 1\} & \text{altrimenti} \end{cases}$$

## KMP

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N];
int b[MAX_N], n, m;
void kmpPreprocess() {
   int i = 0, j = -1; b[0] = -1;
   while (i < m) {
       while (j >= 0 && P[i] != P[j]) j = b[j];
       i++; j++;
       b[i] = j;
   }
}
void kmpSearch() {
   int i = 0, j = 0;
   while (i < n) {
       while (j >= 0 && T[i] != P[j]) j = b[j];
       i++; j++;
       if (j == m) {
           printf("P found at index %d in T\n",
i-j);
           j = b[j];
       }
   }
}
int main() {
   strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY
SEVENTY SEVEN");
   strcpy(P, "SEVENTY SEVEN");
   n = (int)strlen(T);
   m = (int)strlen(P);
   kmpPreprocess();
   kmpSearch();
   return 0;
}
```

# Geometry

## Polygons

```
#define EPS 1e-9
#define PI acos(-1.0)
double DEG_to_RAD(double d) { return d * PI / 180.0;
}
double RAD_to_DEG(double r) { return r * 180.0 / PI;
}

struct point {
   double x, y;
   point() { x = y = 0.0; }
   point(double _x, double _y) : x(_x), y(_y) {}
   bool operator == (point other) const {
       return (fabs(x - other.x) < EPS && (fabs(y - 
other.y) < EPS));
       }
   };

struct vec {
   double x, y;
   vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(point a, point b) {
   return vec(b.x - a.x, b.y - a.y);
}
double dist(point p1, point p2) {
   return hypot(p1.x - p2.x, p1.y - p2.y);
}
double perimeter(const vector<point> &P) {
   double result = 0.0;
   for (int i = 0; i < (int)P.size()-1; i++)
       result += dist(P[i], P[i+1]);
   return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
   double result = 0.0, x1, y1, x2, y2;
   for (int i = 0; i < (int)P.size()-1; i++) {
       x1 = P[i].x; x2 = P[i+1].x;
       y1 = P[i].y; y2 = P[i+1].y;
       result += (x1 * y2 - x2 * y1);
   }
   return fabs(result) / 2.0;
}
double dot(vec a, vec b) {
   return (a.x * b.x + a.y * b.y);
}
double norm_sq(vec v) {
   return v.x * v.x + v.y * v.y;
}
// returns angle aob in rad
double angle(point a, point o, point b) {
 vec oa = toVec(o, a), ob = toVec(o, b);
 return acos(dot(oa, ob) / sqrt(norm_sq(oa) *
norm_sq(ob)));
}
double cross(vec a, vec b) {
   return a.x * b.y - a.y * b.x;
}
// note: to accept collinear points, we have to
change the `> 0'
// returns true if point r is on the left side of
line pq
bool ccw(point p, point q, point r) {
   return cross(toVec(p, q), toVec(p, r)) > 0;
}
// returns true if point r is on the same line as
the line pq
bool collinear(point p, point q, point r) {
 return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}
// returns true if we always make the same turn
while examining
// all the edges of the polygon one by one
```

```
bool isConvex(const vector<point> &P) {
   int sz = (int)P.size();
   if (sz <= 3) return false;
   bool isLeft = ccw(P[0], P[1], P[2]);
   for (int i = 1; i < sz-1; i++)
       if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 :
i+2]) != isLeft)
           return false;
   return true;
}
// returns true if point p is in either
convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
   if ((int)P.size() == 0) return false;
   // assume the first vertex is equal to the last
vertex
   double sum = 0;
   for (int i = 0; i < (int)P.size()-1; i++) {
       if (ccw(pt, P[i], P[i+1]))
           sum += angle(P[i], pt, P[i+1]);
       else
           sum -= angle(P[i], pt, P[i+1]);
   }
   return fabs(fabs(sum) - 2*PI) < EPS;
}

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A,
point B) {
   double a = B.y - A.y;
   double b = A.x - B.x;
   double c = B.x * A.y - A.x * B.y;
   double u = fabs(a * p.x + b * p.y + c);
   double v = fabs(a * q.x + b * q.y + c);
   return point((p.x * v + q.x * u) / (u+v),
       (p.y * v + q.y * u) / (u+v));
}
// cuts polygon Q along the line formed by point a
-> point b
// returns the left side
// (note: the last point must be the same as the
first point)
vector<point> cutPolygon(point a, point b, const
vector<point> &Q) {
   vector<point> P;
   for (int i = 0; i < (int)Q.size(); i++) {
       double left1 = cross(toVec(a, b),
       toVec(a, Q[i])), left2 = 0;
       if (i != (int)Q.size()-1)
           left2 = cross(toVec(a, b), toVec(a,
Q[i+1]));
       // Q[i] is on the left of ab
       if (left1 > -EPS)
           P.push_back(Q[i]);
       // edge (Q[i], Q[i+1]) crosses line ab
       if (left1 * left2 < -EPS)
           P.push_back(lineIntersectSeg(Q[i],
Q[i+1], a, b));
   }
   // make P's first point = P's last point
   if (!P.empty() && !(P.back() == P.front()))
       P.push_back(P.front());
   return P;
}

point pivot;
bool angleCmp(point a, point b) {
   if (collinear(pivot, a, b))
       return dist(pivot, a) < dist(pivot, b);
   double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
   double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
   return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
}

vector<point> CH(vector<point> P) {
   int i, j, n = (int)P.size();
   if (n <= 3) {
       if (!(P[0] == P[n-1]))
           P.push_back(P[0]);
       return P;
   }
   // first, find P0 = point with lowest Y and if
tie: rightmost X
   int P0 = 0;
   for (i = 1; i < n; i++)
       if (P[i].y < P[P0].y ||
           (P[i].y == P[P0].y && P[i].x > P[P0].x))
       P0 = i;
  // swap P[P0] with P[0]
   point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
   // second, sort points by angle w.r.t. pivot P0
   pivot = P[0];
   sort(++P.begin(), P.end(), angleCmp);
   // third, the ccw tests
   vector<point> S;
   S.push_back(P[n-1]);
   S.push_back(P[0]);
   S.push_back(P[1]);
   i = 2;
   while (i < n) {
       j = (int)S.size()-1;
       if (ccw(S[j-1], S[j], P[i]))
           S.push_back(P[i++]);  // left turn,
accept
       else S.pop_back();
   }
   return S;
}
// Area of triangle
double area(double ab, double bc, double ca) {
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) *
sqrt(s - ca); }
```

## Points and lines

```
bool areParallel(line l1, line l2) {
   return (fabs(l1.a-l2.a) < EPS) &&
(fabs(l1.b-l2.b) < EPS);
}
bool areSame(line l1, line l2) {           // also
check coefficient c
 return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) <
```

```
EPS);
}


// returns true (+ intersection point) if two lines
are intersect
bool areIntersect(line l1, line l2, point &p) {
   if (areParallel(l1, l2)) return false;
   p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b
- l1.a * l2.b);
   if (fabs(l1.b) > EPS)
       p.y = -(l1.a * p.x + l1.c);
   else
       p.y = -(l2.a * p.x + l2.c);
   return true;
}


// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
   l.a = -m;
   l.b = 1;
   l.c = -((l.a * p.x) + (l.b * p.y));
}

void closestPoint(line l, point p, point &ans) {
   line perpendicular;
   // vertical line
   if (fabs(l.b) < EPS) {
       ans.x = -(l.c);
       ans.y = p.y;
       return;
   }
   // horizontal line
   if (fabs(l.a) < EPS) {
       ans.x = p.x;
       ans.y = -(l.c);
       return;
   }
   pointSlopeToLine(p, 1 / l.a, perpendicular);
   areIntersect(l, perpendicular, ans);
}


point translate(point p, vec v) {
   return point(p.x + v.x , p.y + v.y);
}
// returns the distance from p to the line defined
by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter
(byref)
double distToLine(point p, point a, point b, point
&c) {
   // formula: c = a + u * ab
   vec ap = toVec(a, p), ab = toVec(a, b);
   double u = dot(ap, ab) / norm_sq(ab);
   c = translate(a, scale(ab, u));
   return dist(p, c);
}
// returns the distance from p to the line segment
ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter
(byref)
double distToLineSegment(point p, point a, point b,
point &c) {
   vec ap = toVec(a, p), ab = toVec(a, b);
   double u = dot(ap, ab) / norm_sq(ab);
   if (u < 0.0) {
       c = point(a.x, a.y);
       return dist(p, a);
   }
   if (u > 1.0) {
       c = point(b.x, b.y);
       return dist(p, b);
   }
   return distToLine(p, a, b, c);
}
```

## Umbral Decoding

```
struct Rect {
   long top, left, bottom, right;
};
bool inArea(long &x,long &y,long &p,long &q,long
&b){
   // NOTE: long product overflows
   __int128 k = abs(x-p), h = abs(y-q);
   return k*k*k + h*h*h <= b;
}

bool overlap(Rect &a, long &x, long &y, long &b){
   long test_x = x, test_y = y;
   if(test_x < a.left)
       test_x = a.left;
   if(test_x > a.right)
       test_x = a.right;
   if(test_y < a.top)
       test_y = a.top;
   if(test_y > a.bottom)
       test_y = a.bottom;
   return inArea(test_x, test_y, x, y, b);
}
bool cover(Rect &a, long &x, long &y, long &b){
   return inArea(a.left, a.top, x, y, b) &&
          inArea(a.left, a.bottom, x, y, b) &&
          inArea(a.right, a.bottom, x, y, b) &&
          inArea(a.right, a.top, x, y, b);
}
int main(int argc, char const *argv[]) {
   long n;
   int k;
   scanf("%ld %d", &n, &k);
   vector<long> x(k), y(k), b(k);
   for(int i = 0; i < k; i++){
       scanf("%ld %ld %ld", &x[i], &y[i], &b[i]);
   }
   stack<Rect> s;
   s.push({0, 0, n, n});
   long area = 0;
   while(!s.empty()){
```

```
        Rect a = s.top();
        s.pop();
        bool cv = false, ol = false;
        for(int i = 0; i < k && !ol && !cv; i++)
            if(overlap(a, x[i], y[i], b[i]))
                if(cover(a, x[i], y[i], b[i]))
                    cv = true;
                else
                    ol = true;
        if(ol){
            if ((a.right-a.left) > (a.bottom-a.top))
{
                long mid_x = (a.right + a.left) / 2;

s.push({a.top,a.left,a.bottom,mid_x});
                s.push({a.top,mid_x+1,a.bottom,
a.right});
            } else {
                long mid_y = (a.bottom + a.top) / 2;
                s.push({a.top,a.left,mid_y,a.right});
                s.push({mid_y+1,a.left,a.bottom,
a.right});
            }
        } else if(!cv)

area+=(a.right-a.left+1)*(a.bottom-a.top+1);
    }
    printf("%ld\n", area);
    return 0;
}
```

# Probability

## Dinner bet

Given the numberNof balls, the numberDof balls drawn each round andtheCvalues chosen by each of thetwo players, find the expected numberof rounds their game will last.

SOLUTION 1:

$$p_{i,j,k} = \frac{\binom{S}{i}\binom{A}{j}\binom{B}{k}\binom{N-A-B-S}{D-i-j-k}}{\binom{N}{D}}$$

$$\mathbb{E}_{S,A,B} = \frac{1}{1-p_{0,0,0}}\left(1 + \sum_{i+j+k\neq 0} p_{i,j,k}\,\mathbb{E}_{S-i,A-j,B-k}\right)$$

$$\mathbb{E}_{0,A,0} = \mathbb{E}_{0,0,B} = 0$$

SOLUTION 2:

The game lasts the most when N= 50 and D= 1 and the players have the same key. With 1 number left, the of probability continuing playing after R rounds is (49/50)R= 0.98R. The contribution to the result is R×0.98R, so an estimate for the minimum number of iterations needed is finding R×0.98R<EPS.Considering 1000 rounds was more than enough for a 10−3 epsilon.

## Probability random graph to be connected

$$c_k = (k-1)^k - \sum_{i=2}^{k-2} c_i \binom{k-1}{i-1}(k-i-1)^k$$

## Binomial coefficient

(n, k) = (n-1, k) + (n-1, k-1)

# Mathematics

## Catalan number

Cat(n) counts the number of distinct binary trees with n vertices

Cat(n) counts the number of expressions containing n pairs of parentheses which are correctly matched.

```
unsigned long int binomialCoeff(unsigned int n,
unsigned int k){
    unsigned long int res = 1;
    if (k > n - k)
        k = n - k;

    for (int i = 0; i < k; ++i)  {
        res *= (n - i);
        res /= (i + 1);
    }
    return res;
}
unsigned long int catalan(unsigned int n){
    unsigned long int c = binomialCoeff(2*n, n);
    return c/(n+1);
}
```

## Matrix exponentiation

F(n) = F(n-1) + F(n-2) + F(n-3) n >= 3;

```
#include<bits/stdc++.h>
using namespace std;
void multiply(int a[3][3], int b[3][3]) {
       int mul[3][3];
       for (int i = 0; i < 3; i++) {
              for (int j = 0; j < 3; j++) {
                     mul[i][j] = 0;
                     for (int k = 0; k < 3; k++)
                            mul[i][j] +=
a[i][k]*b[k][j];
              }
       }
       // storing the multiplication result in a[][]
       for (int i=0; i<3; i++)
              for (int j=0; j<3; j++)
                     a[i][j] = mul[i][j];
}
int power(int F[3][3], int n)
{
       int M[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}};
       if (n==1)
              return F[0][0] + F[0][1];

       power(F, n/2);
       multiply(F, F);
       if (n%2 != 0)
              multiply(F, M);
       return F[0][0] + F[0][1] ;
}
int findNthTerm(int n) {
```

```
        int F[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}} ;
        if(n==0)
                return 0;
        if(n==1 || n==2)
                return 1;
        return power(F, n-2);
}
int main() {
int n = 5;
cout << "F(5) is " << findNthTerm(n);
return 0;
}
```

## Fibonacci

```
// with O(Log n) arithmatic operations
#include <bits/stdc++.h>
using namespace std;
const int MAX = 1000;
int f[MAX] = {0};
int fib(int n) {
        if (n == 0)
                return 0;
        if (n == 1 || n == 2)
                return (f[n] = 1);
        if (f[n])
                return f[n];

        int k = (n & 1)? (n+1)/2 : n/2;
f[n] = (n & 1)? (fib(k)*fib(k) + fib(k-1)*fib(k-1))
                : (2*fib(k-1) + fib(k))*fib(k);
        return f[n];
}
int main() {
        int n = 9;
        printf("%d ", fib(n));
        return 0;
}

O(1) fibonacci
int fib(int n) {
  double phi = (1 + sqrt(5)) / 2;
  return round(pow(phi, n) / sqrt(5));
}
```

## mcd

```
int mcd(int a, int b) {
  int t;
  while (b != 0) {
    t = b;
    b = a % b;
    a = t;
  }
  return a;
}
```

## mcm

```
int mcm(int a, int b){
    return (a*b) / mcd(a,b);
}
```

## Chinese remainder

We are given two arrays num[0..k-1] and rem[0..k-1]. In num[0..k-1], every pair is coprime (gcd for every pair is 1). We need to find minimum positive number x such that:

x % num[0] = rem[0],
x % num[1] = rem[1],
.......................
x % num[k-1] = rem[k-1]

```
#include<bits/stdc++.h>
using namespace std;
int inv(int a, int m) {
   int m0 = m, t, q;
   int x0 = 0, x1 = 1;
   if (m == 1)
       return 0;
   while (a > 1) {
       q = a / m;
       t = m;
       m = a % m, a = t;
       t = x0;
       x0 = x1 - q * x0;
       x1 = t;
   }
   if (x1 < 0)
       x1 += m0;
   return x1;
}

// k is size of num[] and rem[]. Returns the
smallest
// number x such that:
// x % num[0] = rem[0],
// x % num[1] = rem[1],
// ..................
// x % num[k-1] = rem[k-1]
// Assumption: Numbers in num[] are pairwise coprime
// (gcd for every pair is 1)
int findMinX(int num[], int rem[], int k) {
   int prod = 1;
   for (int i = 0; i < k; i++)
       prod *= num[i];
   int result = 0;
   for (int i = 0; i < k; i++){
       int pp = prod / num[i];
       result += rem[i] * inv(pp, num[i]) * pp;
   }
   return result % prod;
}

int main(void) {
   int num[] = {3, 4, 5};
   int rem[] = {2, 3, 1};
   int k = sizeof(num)/sizeof(num[0]);
   cout << "x is " << findMinX(num, rem, k);
   return 0;
}
```

## Gauss Elimination

```
#include <cmath>
#include <cstdio>
```

```
using namespace std;

// adjust this value as needed
#define MAX_N 3
struct AugmentedMatrix { double mat[MAX_N][MAX_N +
1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N,
AugmentedMatrix Aug) {
   // input: N, Augmented Matrix Aug, output: Column
vector X, the answer
   int i, j, k, l; double t;

   for (i = 0; i < N - 1; i++) {
       l = i;
       for (j = i + 1; j < N; j++)
           if (fabs(Aug.mat[j][i]) >
fabs(Aug.mat[l][i]))
               l = j;
       for (k = i; k <= N; k++){
           t = Aug.mat[i][k];
           Aug.mat[i][k] = Aug.mat[l][k];
           Aug.mat[l][k] = t;
       }
       for (j = i + 1; j < N; j++)
           for (k = N; k >= i; k--)
               Aug.mat[j][k] -= Aug.mat[i][k] *
Aug.mat[j][i] / Aug.mat[i][i];
   }

   ColumnVector Ans;
   for (j = N - 1; j >= 0; j--) {
       for (t = 0.0, k = j + 1; k < N; k++)
           t += Aug.mat[j][k] * Ans.vec[k];
       // the answer is here
       Ans.vec[j] = (Aug.mat[j][N] - t) /
Aug.mat[j][j];
   }
   return Ans;
}

// aX + bY + cZ = D
int main() {
   AugmentedMatrix Aug;
   Aug.mat[0][0] = 1;
   Aug.mat[0][1] = 1;
   Aug.mat[0][2] = 2;
   Aug.mat[0][3] = 9;

   Aug.mat[1][0] = 2;
   Aug.mat[1][1] = 4;
   Aug.mat[1][2] = -3;
   Aug.mat[1][3] = 1;

   Aug.mat[2][0] = 3;
   Aug.mat[2][1] = 6;
   Aug.mat[2][2] = -5;
   Aug.mat[2][3] = 0;

   ColumnVector X = GaussianElimination(3, Aug);
   printf("X = %.1lf, Y = %.1lf, Z = %.1lf\n",
X.vec[0], X.vec[1], X.vec[2]);
   return 0;
}
```

## Miller Rabin

```
int power(int x, unsigned int y, int p) {
        int res = 1;
        x = x % p;
        while (y > 0) {
                if (y & 1)
                        res = (res*x) % p;
                y = y>>1;
                x = (x*x) % p;
        }
        return res;
}

bool miillerTest(int d, int n) {
        int a = 2 + rand() % (n - 4);
        int x = power(a, d, n);

        if (x == 1 || x == n-1)
        return true;
        while (d != n-1) {
                x = (x * x) % n;
                d *= 2;
                if (x == 1)     return false;
                if (x == n-1) return true;
        }
        return false;
}
// False if n is composite.
// true if n is probably prime.
// k is the accuracy level.
bool isPrime(int n, int k) {
        if (n <= 1 || n == 4) return false;
        if (n <= 3) return true;
        int d = n - 1;
        while (d % 2 == 0)
                d /= 2;
        for (int i = 0; i < k; i++)
                if (!miillerTest(d, n))
                        return false;

        return true;
}
int main() {
        int k = 4, n = 7;
        cout << isPrime(11, k); //true
        cout << isPrime(6, k); //false
        return 0; }
```

## Median

```
int partition(int arr[], int l, int r, int k);
int findMedian(int arr[], int n) {
        sort(arr, arr+n);
        return arr[n/2];
}
int kthSmallest(int arr[], int l, int r, int k) {
        if (k > 0 && k <= r - l + 1) {
```

```
            int n = r-l+1;
               int i, median[(n+4)/5];
               for (i=0; i<n/5; i++)
                 median[i]=findMedian(arr+l+i*5, 5);
               if (i*5 < n) {

median[i]=findMedian(arr+l+i*5,n%5);
                    i++;
               }
               int medOfMed = (i == 1)? median[i-1]:
                    kthSmallest(median, 0, i-1,
i/2);
               int pos=partition(arr,l,r,medOfMed);
               if (pos-l == k-1)
                   return arr[pos];
               if (pos-l > k-1)
                   return
kthSmallest(arr,l,pos-1,k);
               return kthSmallest(arr,pos+1,r,
k-pos+l-1);
        }
        return INT_MAX;
}
void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp; }
int partition(int arr[], int l, int r, int x) {
        int i;
        for (i=l; i<r; i++)
               if (arr[i] == x)
               break;
        swap(&arr[i], &arr[r]);

        i = l;
        for (int j = l; j <= r - 1; j++) {
               if (arr[j] <= x) {
                      swap(&arr[i], &arr[j]);
                      i++;
               }
        }
        swap(&arr[i], &arr[r]);
        return i;
}
int main() {
        int arr[] = {12, 3, 5, 7, 4, 19, 26};
        int n = sizeof(arr)/sizeof(arr[0]), k = 3;
        cout << "K'th smallest element is "
               << kthSmallest(arr, 0, n-1, k);
        return 0;
}
```

# Data Structures

## Sparse Table

```
#define MAX_N 1000                          //
adjust this value as needed
#define LOG_TWO_N 10            // 2^10 > 1000,
adjust this value as needed

struct RMQ {
// Range Minimum Query
   int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
   RMQ(int n, int A[]) {
       for (int i = 0; i < n; i++) {
           _A[i] = A[i];
           // RMQ of sub array starting at
           // index i + length 2^0=1
           SpT[i][0] = i;
       }
       // complexity = O(n log n)
       for (int j = 1; (1<<j) <= n; j++)
           for (int i = 0; i + (1<<j) - 1 < n; i++)
               if (_A[SpT[i][j-1]] <
_A[SpT[i+(1<<(j-1))][j-1]])
                   SpT[i][j] = SpT[i][j-1];
               else
                   SpT[i][j] =
SpT[i+(1<<(j-1))][j-1];
   }
   int query(int i, int j) {
       int k = (int)floor(log((double)j-i+1) /
log(2.0));
       if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]])
           return SpT[i][k];
       else
           return SpT[j-(1<<k)+1][k];
   }
};
int main() {
   int n = 7, A[] = {18, 17, 13, 19, 15, 11, 20};
   RMQ rmq(n, A);
   for (int i = 0; i < n; i++)
       for (int j = i; j < n; j++)
           printf("RMQ(%d, %d) = %d\n", i, j,
rmq.query(i, j));
   return 0;
}
```

## LCA

```
int L[2*MAX_N], E[2*MAX_N], H[MAX_N], idx;
void dfs(int cur, int depth) {
   H[cur] = idx;
   E[idx] = cur;
   L[idx++] = depth;
   for (int i = 0; i < children[cur].size(); i++) {
       dfs(children[cur][i], depth+1);
       E[idx] = cur;
       L[idx++] = depth;
   }
}
void buildRMQ() {
   idx = 0;
   memset(H, -1, sizeof H);
   dfs(0, 0);
}

int main() {
   // build here rooted tree children (root 0)
   buildRMQ();
   return 0;
}
```

## Fenwick Tree

```
define lsb(x) (x & (-x))
#define MAXN 5000010
long ft[MAXN+1];
long prefix_sum(size_t k) {
   long ans = 0;
   for (; k > 0; k -= lsb(k))
       ans += ft[k];
   return ans;
}
long sum(size_t a, size_t b) {
   return prefix_sum(b) - prefix_sum(a - 1);
}
void update(size_t k, int delta) {
   for (; k <= MAXN; k += lsb(k))
       ft[k] += delta;
}
```

# CPP utils

## String words loop

```
#include <sstream>
string str("abc:def");
char split_char = ':';
istringstream split(str);
vector<string> tokens;
for (string each; getline(split, each, split_char);
tokens.push_back(each));
// now use `tokens`
```

## Priority queue

```
Struct Compare {
   bool operator() (int a, int b) {
       return a > b; // min priority
   }
};
int main(){
   priority_queue<int, vector<int>, Compare> pq;
   return 0;
}
```