Rough Index


```
g++ -std=c++17 -Wshadow -Wall -o "%e" "%f" -O2
-Wno-unused-result

g++ -std=c++17 -Wshadow -Wall -o "%e" "%f" -g
-fsanitize=address -fsanitize=undefined
-D_GLIBCXX_DEBUG
```

ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL)

--- 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[]) {    // constructor as
well as pre-processing routine
   for (int i = 0; i < n; i++) {
     _A[i] = A[i];
     SpT[i][0] = i; // RMQ of sub array
starting at index i + length 2^0=1
   }
   // the two nested loops below have
overall time complexity = O(n log n)
   for (int j = 1; (1<<j) <= n; j++) // for
each j s.t. 2^j <= n, O(log n)
     for (int i = 0; i + (1<<j) - 1 < n;
i++)    // for each valid i, O(n)
       if (_A[SpT[i][j-1]] <
_A[SpT[i+(1<<(j-1))][j-1]])           //
RMQ
         SpT[i][j] = SpT[i][j-1];    //
start at index i of length 2^(j-1)
       else                    // start at
index i+2^(j-1) of length 2^(j-1)
         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));    // 2^k <= (j-i+1)
   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() {
  // same example as in chapter 2: segment tree
  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;
}
```

--- Fenwick Tree // BIT

```
struct Fenwick
{
    vector<ll> t;
    Fenwick(int n){
        t.assign(n+1,0);
    }
    void reset(int n){
            t.assign(n+1, 0);
    }
    void update(int p, ll v){
        for (; p < (int)t.size(); p += (p&(-p))) t[p] += v;
    }
    ll query(int r){  //finds [1, r] sum
        ll sum = 0;
        for (; r; r -= (r&(-r))) sum += t[r];
        return sum;
    }
    ll query(int l, int r){ //finds [l, r] sum
            if(l == 0) return query(r);
            return query(r) - query(l-1);
    }
};

struct FenwickRange{
    vector<ll> fw, fw2;
    int siz;
    FenwickRange(int N){
        fw.assign(N+1,0);
        fw2.assign(N+1,0);
        siz = N+1;
    }
    void reset(int N){
            fw.assign(N+1,0);
        fw2.assign(N+1,0);
        siz = N+1;
    }
    void update(int l, int r, ll val){ //[l, r] + val
        for (int tl = l; tl < siz; tl += (tl&(-tl))){
            fw[tl] += val, fw2[tl] -= val * ll(l - 1);
        }
        for (int tr = r + 1; tr < siz; tr += (tr&(-tr))){
            fw[tr] -= val, fw2[tr] += val * ll(r);
        }
    }
    ll sum(int r){ //[1, r]
        ll res = 0;
        for (int tr = r; tr; tr -= (tr&(-tr))){
            res += fw[tr] * ll(r) + fw2[tr];
        }
        return res;
    }
    ll query(ll l, ll r){
            if(l == 0) return sum(r);
            else return sum(r)-sum(l-1);
    }
};

struct Fenwick2D{
    int R, C;
    vector< vector<ll> > fw;
    Fenwick2D(int r, int c) {
        R = r; C = c;
        fw.assign(R+1, vector<ll>(C+1,0));
    }
    void reset(int r, int c){
            R = r; C = c;
        fw.assign(R+1, vector<ll>(C+1,0));
    }
    void update(int row, int col, ll val) {
        for (int r = row; r < R; r += (r&(-r))){
            for(int c = col; c < C; c += (c&(-c))) {
                fw[r][c] += val;
            }
        }
    }
    ll sum(int row, int col){   // inclusive
        ll res = 0;
        for (int r = row; r; r -= (r&(-r))){
            for(int c = col; c; c -= (c&(-c))) {
                res += fw[r][c];
            }
        }
        return res;
    }
    ll query(int x1, int y1, int x2, int y2){
        return sum(x2, y2) - sum(x1-1, y2) - sum(x2, y1-1) +
sum(x1-1, y1-1);
    }
};
```

--- Segment Tree

```
const int mxn = 2e5+5;
int64_t a[mxn];

struct ST{
        vector<int64_t> tree,lazy;
        void init(int n){
                tree.assign(n*4,0);
                lazy.assign(n*4,0);
                build(1,0,n-1);
```

```
}
void build(int id, int le, int ri){
	if(le==ri){
		tree[id]=a[le];
		return;
	}
	int mid = (le+ri)>>1;
	build(id*2,le,mid);
	build(id*2+1,mid+1,ri);
	pushup(id,le,ri);
}
void puttag(int id, int le, int ri, int64_t v){
	int len = ri-le+1;
	tree[id]+=v*len;
	lazy[id]+=v;
}
void pushdown(int id, int le, int ri){
	int64_t x = lazy[id];
	lazy[id]=0;
	if(x==0||le==ri){
		return;
	}
	int mid=(le+ri)>>1;
	puttag(id*2,le,mid,x);
	puttag(id*2+1,mid+1,ri,x);
}
void pushup(int id, int le, int ri){
	tree[id]=tree[id*2]+tree[id*2+1];
}
//update by point
void upd1(int x, int v, int id, int le, int ri){
	if(le==ri){
		puttag(id,le,ri,v);
		return;
	}
	pushdown(id,le,ri);
	int mid=(le+ri)>>1;
	if(x<=mid)upd1(x,v,id*2,le,mid);
	else upd1(x,v,id*2+1,mid+1,ri);
	pushup(id,le,ri);
}

//update by range
void upd2(int x, int y, int64_t v, int id, int le, int ri){
	if(x>ri||le>y)return;
	if(x<=le&&ri<=y){
		puttag(id,le,ri,v);
		return;
	}
	pushdown(id,le,ri);
	int mid=(le+ri)>>1;
	upd2(x,y,v,id*2,le,mid);
		upd2(x,y,v,id*2+1,mid+1,ri);
		pushup(id,le,ri);
	}
	int64_t query(int x, int y, int id, int le, int ri){
		if(x>ri||le>y)return 0;
		if(x<=le&&ri<=y){
			return tree[id];
		}
		pushdown(id,le,ri);
		int mid=(le+ri)>>1;
		int64_t left = query(x,y,id*2,le,mid);
		int64_t right = query(x,y,id*2+1,mid+1,ri);
		return left+right;
	}
};
```

--- PBDS

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

using namespace std;
using namespace __gnu_pbds;

template<class T> using oset =
tree<T,null_type,less<T>,rb_tree_tag,tree_order_statistic
s_node_update>;

int main() {
   oset<ii>t;
   t.insert(ii(1,3));
   t.insert(ii(5,4));
   cout<<t.order_of_key(ii(3,2))<<'\n'; //1
   ii tmp = *(t.find_by_order(0)); //ii(1,3)
   return 0;
}
```

--- Articulation Points & Bridges

```
vector<int> dfs_low, articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      //
dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {                    //
a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case,
count children of root
```

```
    articulationPointAndBridge(v.first);

    if (dfs_low[v.first] >= dfs_num[u])             // for
articulation point
      articulation_vertex[u] = true;          // store this
information first
    if (dfs_low[v.first] > dfs_num[u])                        // for
bridge
      printf(" Edge (%d, %d) is a bridge\n", u, v.first);
    dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);      //
update dfs_low[u]
  }
  else if (v.first != dfs_parent[u])      // a back edge and
not direct cycle
    dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);      //
update dfs_low[u]
} }

// inside int main()
printThis("Articulation Points & Bridges (the input graph
must be UNDIRECTED)");
dfsNumberCounter = 0; dfs_num.assign(V,
DFS_WHITE); dfs_low.assign(V, 0);
dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; i++)
  if (dfs_num[i] == DFS_WHITE) {
    dfsRoot = i; rootChildren = 0;
    articulationPointAndBridge(i);
    articulation_vertex[dfsRoot] = (rootChildren > 1); }
// special case
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
  if (articulation_vertex[i])
    printf(" Vertex %d\n", i);
```

--- DFS SCC Toposort DAG

```
#include <vector>
#include <stack>

using namespace std;

int height[100100], cur_num;
vector<vector<int> > graph, dag;
vector<int> dag_label, dag_max, dag_min, dfs_num,
dfs_low, topological;
vector<bool> is_parent, visited;
stack<int> s;

void topological_sort(int u){
  visited[u] = true;
  for(int i = 0; i < dag[u].size(); i++){
    int v = dag[u][i];
    if(!visited[v])
      topological_sort(v);
  }
  topological.push_back(u);
}
void scc_dfs(int u){
  dfs_num[u] = dfs_low[u] = cur_num++;
  is_parent[u] = true;
  s.push(u);

  for(int i = 0; i < graph[u].size(); i++){
    int v = graph[u][i];
    if(dfs_num[v] == -1){
      scc_dfs(v);
    }
    // a back edge
    if(is_parent[v]){
      dfs_low[u] = min(dfs_low[u], dfs_low[v]);
    }
  }
  if(dfs_low[u] == dfs_num[u]){
    // root of the scc
    int cur_dag = dag_max.size(), v;
    dag_max.push_back(height[u]);
    dag_min.push_back(height[u]);
    do{
      v = s.top(); s.pop();
      is_parent[v] = false;

      dag_label[v] = cur_dag;
      dag_max[cur_dag] = max(dag_max[cur_dag],
height[v]);
      dag_min[cur_dag] = min(dag_min[cur_dag],
height[v]);
    }while(u != v);
  }
}
int main(){
  int t;
  cin >> t;

  while(t--){
    int n, m;
    cin >> n >> m;

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

    graph.assign(n+1, vector<int> ());
```

```
    while(m--){
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
    }

    // find all scc
    dag_max.clear();
    dag_min.clear();
    dfs_num.assign(n+1, -1);
    dfs_low.assign(n+1, -1);
    dag_label.assign(n+1, -1);
    is_parent.assign(n+1, false);
    cur_num = 1;
    for(int i = 1; i <= n; i++){
        if(dfs_num[i] == -1)
            scc_dfs(i);
    }

    // build dag, dont care repeated edge,
    // just repeat it, not going to harm u in anyway
    dag.assign(dag_max.size(), vector<int> ());
    for(int i = 1; i <= n; i++){
        int u = dag_label[i];
        for(int j = 0; j < graph[i].size(); j++){
            int v = dag_label[graph[i][j]];

            if(u == v) continue;
            dag[u].push_back(v);
        }
    }

    topological.clear();
    visited.assign(dag_max.size(), false);
    for(int i = 0; i < dag_max.size(); i++){
        if(!visited[i])
            topological_sort(i);
    }

    int ans = 0;
    for(vector<int>::iterator it = topological.begin(); it !=
topological.end(); it++){
        int u = *it;
        for(int i = 0; i < dag[u].size(); i++){
            int v = dag[u][i];
            dag_max[u] = max(dag_max[u], dag_max[v]);
        }
        ans = max(ans, dag_max[u] - dag_min[u]);
    }
    cout << ans << endl;
  }
  return 0;
}
```

--- SPFA

```
#include <vector>
#include <queue>
using namespace std;
typedef pair<int, int> ii;
vector<vector<ii> > graph;
int main() {
  int tc;
  cin >> tc;
  while(tc--){
    int n, m;
    cin >> n >> m;
    graph.assign(n, vector<ii> () );
    while(m--){
      int a, b, t;
      cin >> a >> b >> t;
      graph[a].push_back(ii(b, t));
    }
    vector<int> dist(n, 1<<30);     dist[0] = 0;
    vector<int> queuetime(n, 0);    queuetime[0] = 1;
    vector<bool> inqueue(n, 0);     inqueue[0] = true;
    queue<int> q;                   q.push(0);
    bool negativecycle = false;

    while(!q.empty() && !negativecycle){
      int u = q.front(); q.pop();
      inqueue[u] = false;

      for(int i = 0; i < graph[u].size(); i++){
        int v = graph[u][i].first, w = graph[u][i].second;

        if(dist[u] + w < dist[v]){
          dist[v] = dist[u] + w;

          if(!inqueue[v]){
            q.push(v);
            queuetime[v]++;
            inqueue[v] = true;

            if(queuetime[v] == n+2){
              negativecycle = true;
              break;
    }  }  }  }  }
    cout << (negativecycle?"possible":"not possible") <<
endl;
  }
  return 0;
}
```

--- HLD

```
// change segment tree merge, query merge
#include <cstring>
#include <cstdio>
#include <vector>

using namespace std;

// change this
#define MAXN 100100
#define LN   20

#define l_node  ( 2*node )
#define r_node  ( 2*node + 1 )
#define mid     ( (l+r)/2 )

typedef pair<int, int> ii;

int n;
vector<vector<ii> > graph;
int parent[MAXN][LN], subtree[MAXN], depth[MAXN],
specialChild[MAXN], chainId[MAXN],
edgePosInChain[MAXN];

void init(int node, int l, int r, vector<int>& st, vector<int>&
a);
void update(int node, int l, int r, int x, int y, long long val,
vector<int>& st, vector<int>& lazy);
int query(int node, int l, int r, int x, int y, vector<int>& st,
vector<int>& lazy);

struct Chain{
   int connect, id;
   vector<int> chainEdge, st, lazy;

   Chain(int u, int _id){
      id = _id;
      connect = u;
   }

   void insert(int v, int w){
      edgePosInChain[v] = chainEdge.size();
      chainEdge.push_back(w);
      chainId[v] = id;
   }

   void initST(){
      st.assign(4*chainEdge.size(), 0);
      lazy.assign(4*chainEdge.size(), 0);

      init(1, 0, chainEdge.size()-1, st, chainEdge);
   }

   int queryChain(int v){
      return query(1, 0, chainEdge.size()-1, 0,
edgePosInChain[v], st, lazy);
   }

   int queryChain(int u, int v){
      return query(1, 0, chainEdge.size()-1,
edgePosInChain[u]+1, edgePosInChain[v], st, lazy);
   }

   void updateChain(int v, int w){
      int pos = edgePosInChain[v];
      chainEdge[pos] = w;
      update(1, 0, chainEdge.size()-1, pos, pos, w, st,
lazy);
   }

};

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

   specialChild[u] = -1;
   for(int i = 0; i < graph[u].size(); i++){
      int v = graph[u][i].first;

      if(v == p) continue;
      dfs(v, u, d+1);
      subtree[u] += subtree[v];

      if(specialChild[u] == -1 || subtree[v] >
subtree[specialChild[u]])
         specialChild[u] = v;
   }
}

vector<Chain*> hldChain;
void HLD(int u, int u_w, Chain* chain){
   chain->insert(u, u_w);

   for(int i = 0; i < graph[u].size(); i++){
      int v = graph[u][i].first, w = graph[u][i].second;

      if(v == parent[u][0]) continue;
      if(v == specialChild[u]){
         // extend chain
         HLD(v, w, chain);
      }
      else{
         // new chain
```

```
            hldChain.push_back(new Chain(u, 
hldChain.size()));
            HLD(v, w, hldChain.back());
        }
    }
}

void HLD(int root){
    memset(parent, -1, sizeof parent);

    dfs(root, -1, 0);
    for(int i = 0; i < LN-1; i++){
        for(int j = 0; j < n; j++){
            if(parent[j][i] != -1)
                parent[j][i+1] = parent[parent[j][i]][i];
        }
    }
    hldChain.push_back(new Chain(-1, hldChain.size()));
    HLD(root, -1, hldChain.back());

    for(int i = 0; i < hldChain.size(); i++)
        hldChain[i]->initST();
}

// each edge has 1-1 correspondent with a vertex except 
root
// v-par[v] edge is uniquely determine by v
void update(int v, int val){
    hldChain[chainId[v]]->updateChain(v, val);
}

int lca(int u, int v){
    if(depth[u] < depth[v])
        return lca(v, u);
    // u is deeper
    int diff = depth[u] - depth[v];

    // advance u with diff
    for(int bit = 0; bit < LN; bit++)
        if(diff & (1<<bit) ) // if ith bit is 1, advance
            u = parent[u][bit];
    if(u != v)
    {
        for(int power = LN-1; power >= 0; power--) // start 
with highest power of 2
            if(parent[u][power] != parent[v][power])    // find 
higest not same parent
            {
                u = parent[u][power];
                v = parent[v][power];
            }
        u = parent[u][0];
    }
    return u;
}

int queryHLD(int u, int v){
    int ans = 0;
    while(chainId[u] != chainId[v]){
        ans = max(hldChain[chainId[u]]->queryChain(u), 
ans);
        u = hldChain[chainId[u]]->connect;
    }

    return max(ans, hldChain[chainId[u]]->queryChain(v, 
u));
}

int query(int u, int v){
    int l = lca(u, v);

    int q1 = queryHLD(u, l);
    int q2 = queryHLD(v, l);

    return max(q1, q2);
}

int main(){
    int t;
    scanf("%d ", &t);

    while(t--){
        scanf("%d", &n);

        vector<ii> edge;
        graph.assign(n, vector<ii>());
        for(int i = 0; i < n-1; i++){
            int u, v, c;
            scanf("%d %d %d", &u, &v, &c);
            u--; v--;
            graph[u].push_back(ii(v, c));
            graph[v].push_back(ii(u, c));
            edge.push_back(ii(u, v));
        }

        HLD(0);

        char type[10];
        scanf("%s", type);
        while(type[0] != 'D'){
            int a, b;
            scanf("%d %d", &a, &b);

            if(type[0]=='Q'){
```

```
            printf("%d\n", query(a-1, b-1));
        }
        else{
            int u = edge[a-1].first, v = edge[a-1].second;

            update(depth[u]>depth[v]?u:v, b);
        }

        scanf("%s", type);
      }
   }

   return 0;
}
```

// segment tree

```
void init(int node, int l, int r, vector<int>& st, vector<int>&
a){
   if(l == r){
      st[node] = a[l];
      return;
   }
   init(l_node, l, mid, st, a);
   init(r_node, mid+1, r, st, a);
   st[node] = max(st[l_node], st[r_node]);
}

void update(int node, int l, int r, int x, int y, long long val,
vector<int>& st, vector<int>& lazy){
   if(lazy[node] != 0){
      // update this node
      st[node] += lazy[node]*(r-l+1);
      if(l != r){
         // propogate down
         lazy[l_node] += lazy[node];
         lazy[r_node] += lazy[node];
      }
      lazy[node] = 0;
   }
   if(x <= l && r <= y){   // in range, lazy update it
      // ensure this node value is correct for parent
updating
      // st[node] += val*(r-l+1);

      // if(l != r){
      //     lazy[l_node] += val;
      //     lazy[r_node] += val;
      // }
      st[node] = val;
      return;
   }
   if(y < l || x > r){
      // out of range
      return;
   }
   update(l_node, l, mid, x, y, val, st, lazy);
   update(r_node, mid+1, r, x, y, val, st, lazy);
   st[node] = max(st[l_node], st[r_node]);
}

int query(int node, int l, int r, int x, int y, vector<int>& st,
vector<int>& lazy){
   if(lazy[node] != 0){
      // update this node
      st[node] += lazy[node]*(r-l+1);
      if(l != r){
         // propogate down
         lazy[l_node] += lazy[node];
         lazy[r_node] += lazy[node];
      }
      lazy[node] = 0;
   }
   if(x <= l && r <= y){
      // in range, return value
      return st[node];
   }
   if(y < l || x > r){
      // out of range
      return 0;
   }
   long long q1 = query(l_node, l, mid, x, y, st, lazy);
   long long q2 = query(r_node, mid+1, r, x, y, st, lazy);
   return max(q1, q2);
}
```

--- LCA

```
#include <vector>
#include <cstring>
#include <cstdio>

using namespace std;

typedef pair<int, int> ii;
typedef long long ll;

vector<vector<ii> > graph;
ll dist[100010];
int parent[100010][20];
int depth[100010];

void dfs(int u){
   for(int i = 0; i < graph[u].size(); i++){
      int v = graph[u][i].first, w = graph[u][i].second;
      if(dist[v] == -1){
```

```
        dist[v] = dist[u] + w;
        depth[v] = depth[u] + 1;
        dfs(v);
      }
    }
}

int lca(int u, int v){
  if(depth[u] < depth[v])
    return lca(v, u);

  // u is deeper

  int diff = depth[u] - depth[v];

  // advance u with diff
  for(int bit = 0; bit < 20; bit++)
    if(diff & (1<<bit) ) // if ith bit is 1, advance
      u = parent[u][bit];

  if(u != v)
  {
    for(int power = 19; power >= 0; power--) // start with
highest power of 2
        if(parent[u][power] != parent[v][power])    // find
higest not same parent
        {
          u = parent[u][power];
          v = parent[v][power];
        }
    u = parent[u][0];
  }
  return u;

}

int main() {
  int N;
  while(scanf("%d", &N) && N != 0){
    graph.assign(N, vector<ii>());
    memset(dist, -1, sizeof dist);
    memset(parent, -1, sizeof parent);

    int v, w;
    for(int i = 1; i < N; i++){
      //cin >> v >> w;
      scanf("%d %d", &v, &w);
      parent[i][0] = v;
      graph[i].push_back(ii(v, w));
      graph[v].push_back(ii(i, w));
    }
    for(int i = 0; i < 19; i++){
      for(int j = 0; j < N; j++){
        if(parent[j][i] != -1)
          parent[j][i+1] = parent[parent[j][i]][i];
    }  }

    dist[0] = 0; depth[0] = 0; dfs(0);

    int Q;
    cin >> Q;
    while(Q--){
      int u, v;
      cin >> u >> v;
      cout << dist[u]+dist[v]-2*dist[lca(u, v)];
      if(!Q) cout << endl;
      else   cout << ' ';
  }   }
  return 0;
}
```

--- Kruskal + UFDS

```
#include <cstdio>
#include <vector>
#include <queue>

using namespace std;

typedef pair<int, int> ii;
typedef pair<int, ii> iii;

vector<int> p;

void init(int n){
  p.resize(n);
  for(int i = 0; i < n; i++){
    p[i] = i;
  }
}
int find_set(int x){
  if(p[x] == x)
    return x;
  return p[x] = find_set(p[x]);
}
bool is_same_set(int x, int y){
  return find_set(x) == find_set(y);
}
void union_set(int x, int y){
  p[find_set(x)] = find_set(y);
}
int main(){
  int n, m, a, b, w;
  cin >> n >> m;
```

```
    init(n);
    priority_queue<iii, vector<iii>, greater<iii> > pq;
    while(m--){
        cin >> a >> b >> w;
        pq.push(iii(w, ii(a-1, b-1)));
    }
    int mst = 0;
    while(!pq.empty()){
        int w = pq.top().first;
        ii v = pq.top().second;
        pq.pop();

        if(!is_same_set(v.first, v.second)){
            mst += w;
            union_set(v.first, v.second);
        }
    }
    cout << mst << endl;

    return 0;
}
```

--- Dinic

```
#include <vector>
#include <map>
#include <cstdio>

#define INF     (1<<30)
#define INFLL   (1LL<<60)

using namespace std;

typedef pair<int, int> ii;

struct MaxFlow{
    int n, s, t;
    vector<vector<ii> > graph;
    vector<long long> cap;
    vector<int> dist, q, now;

    MaxFlow(int _n){
        // 0-based indexing, init(n+1) for 1 based indexing
        n = _n;
        graph.assign(n, vector<ii> ());
        q.resize(n+10);
    }

    void addEdge(int u, int v, long long c, bool directed){
        graph[u].push_back(ii(v, cap.size()));
        cap.push_back(c);
        graph[v].push_back(ii(u, cap.size()));
        cap.push_back(directed?0:c);
    }

    long long getMaxFlow(int _s, int _t){
        s = _s; t = _t;
        long long flow = 0;
        while(bfsLevelGraph()){
            now.assign(n, 0);
            while(long long f = dfsSendFlow(s, INFLL))
                flow += f;
        }

        return flow;
    }

    bool bfsLevelGraph(){
        dist.assign(n, INF);
        int qs = 0, qe = 0;
        q[qe++] = s;
        dist[s] = 0;

        while(qs < qe){
            int u = q[qs++];
            for(int i = 0; i < graph[u].size(); i++){
                int v = graph[u][i].first, e = graph[u][i].second;

                if(dist[v] == INF && cap[e] > 0){
                    dist[v] = dist[u]+1;
                    q[qe++] = v;
        }   }   }
        return dist[t] != INF;
    }
    long long dfsSendFlow(int u, long long curFlow){
        if(u == t)          return curFlow;
        if(curFlow == 0)    return curFlow;

        for(; now[u] < graph[u].size(); now[u]++){
            int v = graph[u][now[u]].first, e =
graph[u][now[u]].second;

            if(dist[v] == dist[u] +1 && cap[e] > 0){
                // an edge exist in level graph
                long long flowSent = dfsSendFlow(v,
min(curFlow, cap[e]));

                if(flowSent > 0){
                    cap[e] -= flowSent;
                    cap[e^1] += flowSent;

                    return flowSent;
        }   }   }
        return 0;
    }
```

```
};

int main(){
   int n, m;
   scanf("%d %d", &n, &m);

   MaxFlow flow_problem(n+1);

   while(m--){
      int u, v, c;
      scanf("%d %d %d", &u, &v, &c);

      if(u == v)
         continue;

      flow_problem.addEdge(u, v, c, false);
   }

   printf("%lld\n", flow_problem.getMaxFlow(1, n));

   return 0;
}
```

--- MinCost Flow Dijkstra

```
// not yet tested on starting negative edge

#include <vector>
#include <queue>

using namespace std;

typedef pair<int, int> ii;

struct Edge{
   int u, v;
   long long cap, cost;

   Edge(int _u, int _v, long long _cap, long long _cost){
      u = _u; v = _v; cap = _cap; cost = _cost;
   }
};

struct MinCostFlow{
   int n, s, t;
   long long flow, cost;
   vector<vector<int> > graph;
   vector<Edge> e;
   vector<long long> dist, potential;
   vector<int> parent;
   bool negativeCost;

   MinCostFlow(int _n){
      // 0-based indexing
      n = _n;
      graph.assign(n, vector<int> ());
      negativeCost = false;
   }

   void addEdge(int u, int v, long long cap, long long cost,
bool directed){
      if(cost < 0)
         negativeCost = true;

      graph[u].push_back(e.size());
      e.push_back(Edge(u, v, cap, cost));

      graph[v].push_back(e.size());
      e.push_back(Edge(v, u, 0, -cost));

      if(!directed)
         addEdge(v, u, cap, cost, true);
   }

   pair<long long, long long> getMinCostFlow(int _s, int
_t){
      s = _s; t = _t;
      flow = 0, cost = 0;

      potential.assign(n, 0);
      if(negativeCost){
         // run Bellman-Ford to find starting potential
         dist.assign(n, 1LL<<62);
         for(int i = 0, relax = false; i < n && relax; i++, relax
= false){
            for(int u = 0; u < n; u++){
               for(int k = 0; k < graph[u].size(); k++){
                  int eIdx = graph[u][i];
                  int v = e[eIdx].v, cap = e[eIdx].cap, w =
e[eIdx].cost;

                  if(dist[v] > dist[u] + w && cap > 0){
                     dist[v] = dist[u] + w;
                     relax = true;
         }  }  }  }

         for(int i = 0; i < n; i++){
            if(dist[i] < (1LL<<62)){
               potential[i] = dist[i];
      }  }  }

      while(dijkstra()){
         flow += sendFlow(t, 1LL<<62);
      }
```

```
    return make_pair(flow, cost);
  }

  bool dijkstra(){
    parent.assign(n, -1);
    dist.assign(n, 1LL<<62);
    priority_queue<ii, vector<ii>, greater<ii> > pq;

    dist[s] = 0;
    pq.push(ii(0, s));


    while(!pq.empty()){
      int u = pq.top().second;
      long long d = pq.top().first;
      pq.pop();

      if(d != dist[u]) continue;

      for(int i = 0; i < graph[u].size(); i++){
        int eIdx = graph[u][i];
        int v = e[eIdx].v, cap = e[eIdx].cap;
        int w = e[eIdx].cost + potential[u] - potential[v];

        if(dist[u] + w < dist[v] && cap > 0){
          dist[v] = dist[u] + w;
          parent[v] = eIdx;

          pq.push(ii(dist[v], v));
    }  }  }

    // update potential
    for(int i = 0; i < n; i++){
      if(dist[i] < (1LL<<62))
        potential[i] += dist[i];
    }

    return dist[t] != (1LL<<62);
  }

  long long sendFlow(int v, long long curFlow){
    if(parent[v] == -1)
      return curFlow;
    int eIdx = parent[v];
    int u = e[eIdx].u, w = e[eIdx].cost;

    long long f = sendFlow(u, min(curFlow,
e[eIdx].cap));

    cost += f*w;
    e[eIdx].cap -= f;
    e[eIdx^1].cap += f;

      return f;
    }
};

int main(){
  int n, m, s, t;
  cin >> n >> m >> s >> t;

  MinCostFlow minCostFlowProblem(n);

  while(m--){
    int u, v, c, w;
    cin >> u >> v >> c >> w;

    minCostFlowProblem.addEdge(u, v, c, w, true);
  }

  pair<int, int> ans =
minCostFlowProblem.getMinCostFlow(s, t);

  cout << ans.first << ' ' << ans.second << endl;

  return 0;
}
```

--- MinCost Flow SPFA

```
struct Edge{
  int u, v;
  long long cap, cost;

  Edge(int _u, int _v, long long _cap, long long _cost){
    u = _u; v = _v; cap = _cap; cost = _cost;
  }
};

struct MinCostFlow{
  int n, s, t;
  long long flow, cost;
  vector<vector<int> > graph;
  vector<Edge> e;
  vector<long long> dist;
  vector<int> parent;

  MinCostFlow(int _n){
    // 0-based indexing
    n = _n;
    graph.assign(n, vector<int> ());
  }

  void addEdge(int u, int v, long long cap, long long cost,
bool directed){
```

```
      graph[u].push_back(e.size());
      e.push_back(Edge(u, v, cap, cost));

      graph[v].push_back(e.size());
      e.push_back(Edge(v, u, 0, -cost));

      if(!directed)
         addEdge(v, u, cap, cost, true);
   }

   pair<long long, long long> getMinCostFlow(int _s, int
_t){
      s = _s; t = _t;
      flow = 0, cost = 0;

      while(SPFA()){
         flow += sendFlow(t, 1LL<<62);
      }

      return make_pair(flow, cost);
   }

   // not sure about negative cycle
   bool SPFA(){
      parent.assign(n, -1);
      dist.assign(n, 1LL<<62);       dist[s] = 0;
      vector<int> queuetime(n, 0);   queuetime[s] = 1;
      vector<bool> inqueue(n, 0);    inqueue[s] = true;
      queue<int> q;                  q.push(s);
      bool negativecycle = false;


      while(!q.empty() && !negativecycle){
         int u = q.front(); q.pop(); inqueue[u] = false;

         for(int i = 0; i < graph[u].size(); i++){
            int eIdx = graph[u][i];
            int v = e[eIdx].v, w = e[eIdx].cost, cap =
e[eIdx].cap;

            if(dist[u] + w < dist[v] && cap > 0){
               dist[v] = dist[u] + w;
               parent[v] = eIdx;

               if(!inqueue[v]){
                  q.push(v);
                  queuetime[v]++;
                  inqueue[v] = true;

                  if(queuetime[v] == n+2){
                     negativecycle = true;
                     break;
      }  }  }  }  }

      return dist[t] != (1LL<<62);
   }

   long long sendFlow(int v, long long curFlow){
      if(parent[v] == -1)
         return curFlow;
      int eIdx = parent[v];
      int u = e[eIdx].u, w = e[eIdx].cost;

      long long f = sendFlow(u, min(curFlow,
e[eIdx].cap));

      cost += f*w;
      e[eIdx].cap -= f;
      e[eIdx^1].cap += f;

      return f;
   }
};

int main(){
   int n, m, s, t;
   cin >> n >> m >> s >> t;

   MinCostFlow minCostFlowProblem(n);

   while(m--){
      int u, v, c, w;
      cin >> u >> v >> c >> w;

      minCostFlowProblem.addEdge(u, v, c, w, true);
   }

   pair<int, int> ans =
minCostFlowProblem.getMinCostFlow(s, t);

   cout << ans.first << ' ' << ans.second << endl;

   return 0;
}
```

--- Bipartite Matching

```
#include <vector>
#include <queue>

#define INF (1<<30)

using namespace std;

struct Matching{
```

```
int n, m;
vector<vector<int> > graph;
vector<int> match, dist;


Matching(int _n, int _m){
   // 1-based indexing
   n = _n; m = _m;
   graph.assign(n+m+1, vector<int> ());
   match.assign(n+m+1, 0);
   dist.resize(n+1);
}

void addPair(int u, int v){
   graph[u].push_back(v+n);
   graph[v+n].push_back(u);
}

int HopcroftKarpMatching(){
   int matching = 0;
   while(bfs()){
      for(int i = 1; i <= n; i++){
         if(match[i] == 0 && dfs(i))
            matching++;
      }
   }

   return matching;
}

bool bfs(){
   // 0 is the sink

   queue<int> q;
   dist[0] = INF;
   for(int i = 1; i <= n; i++){
      if(match[i] == 0){
         dist[i] = 0;
         q.push(i);
      }
      else{
         dist[i] = INF;
      }
   }

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

      if(u != 0){
         for(int i = 0; i < graph[u].size(); i++){
            int v = graph[u][i];
            // v is in V, match[v] is in U
            // match[v] is 0 (not matched by default)
            if(dist[match[v]] == INF){
               dist[match[v]] = dist[u] + 1;
               q.push(match[v]);
   }  }  }  }
   // check is sink is still recheable
   return dist[0] != INF;
}

int dfs(int u){
   // reach sink
   if(u == 0)
      return true;

   for(int i = 0; i < graph[u].size(); i++){
      int v = graph[u][i];

      if(dist[match[v]] == dist[u] + 1 && dfs(match[v])){
         match[u] = v;
         match[v] = u;
         return true;
      }
   }
   // no more match available
   return false;
}
};

int main(){
   int n, m, e;
   scanf("%d %d %d", &n, &m, &e);

   Matching matching_problem(n, m);

   while(e--){
      int u, v;
      scanf("%d %d", &u, &v);

      matching_problem.addPair(u, v);
   }

   printf("%d\n",
matching_problem.HopcroftKarpMatching());

   return 0;
}
```

--- Suffix Array

```
typedef pair<int, int> ii;
#define MAX_N 100010 // second approach: O(n log n)
char T[MAX_N]; // the input string, up to 100K characters

int n; // the length of input string
```

```
int RA[MAX_N], tempRA[MAX_N]; // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N]; // suffix array and temporary suffix array
int c[MAX_N]; // for counting/radix sort
char P[MAX_N]; // the pattern string (for string matching)
int m; // the length of pattern string

int Phi[MAX_N]; // for computing longest common prefix
int PLCP[MAX_N];
int LCP[MAX_N]; // LCP[i] stores the LCP between previous suffix T+SA[i-1]
// and current suffix T+SA[i]

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; } // compare

void constructSA_slow() { // cannot go beyond 1000 characters
	for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
	sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log
n)
}

void countingSort(int k) { // O(n)
	int i, sum, maxi = max(300, n); // up to 255 ASCII chars or length of n
	memset(c, 0, sizeof c); // clear frequency table
	for (i = 0; i < n; i++) // count the frequency of each integer rank
		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++) // shuffle the suffix array if necessary
		tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
	for (i = 0; i < n; i++) // update the suffix array SA
		SA[i] = tempSA[i];
}

void constructSA() { // this version can go up to 100000 characters
	int i, k, r;
	for (i = 0; i < n; i++) RA[i] = T[i]; // initial rankings
	for (i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
	for (k = 1; k < n; k <<= 1) { // repeat sorting process log n times
		countingSort(k); // actually radix sort: sort based on the
second item
		countingSort(0); // then (stable) sort based on the first item
		tempRA[SA[0]] = r = 0; // re-ranking; start from rank r = 0
		for (i = 1; i < n; i++) // compare adjacent suffixes
			tempRA[SA[i]] = // if same pair => same rank r;
otherwise, increase r
			(RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] ==
RA[SA[i-1]+k]) ? r : ++r;
		for (i = 0; i < n; i++) // update the rank array RA
			RA[i] = tempRA[i];
		if (RA[SA[n-1]] == n-1) break; // nice optimization trick
} }

void computeLCP_slow() {
	LCP[0] = 0; // default value
	for (int i = 1; i < n; i++) { // compute LCP by definition
	int L = 0; // always reset L to 0
	while (T[SA[i] + L] == T[SA[i-1] + L]) L++; // same L-th char, L++
	LCP[i] = L;
} }

void computeLCP() {
	int i, L;
	Phi[SA[0]] = -1; // default value
	for (i = 1; i < n; i++) // compute Phi in O(n)
		Phi[SA[i]] = SA[i-1]; // remember which suffix is behind this
suffix
	for (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)
		if (Phi[i] == -1) { PLCP[i] = 0; continue; } // special case
		while (T[i + L] == T[Phi[i] + L]) L++; // L increased max n
times
		PLCP[i] = L;
		L = max(L-1, 0); // L decreased max n times
}
	for (i=0; i<n; i++) // compute LCP in O(n)
		LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the
correct position
}

ii stringMatching() { // string matching in O(m log n)
	int lo = 0, hi = n-1, mid = lo; // valid matching = [0..n-1]
	while (lo < hi) { // find lower bound
		mid = (lo + hi) / 2; // this is round down
		int res = strncmp(T + SA[mid], P, m); // try to find P in suffix
'mid'
		if (res >= 0) hi = mid; // prune upper half (notice the >=
sign)
		else lo = mid + 1; // prune lower half including mid
	} // observe `=' in "res >= 0" above
	if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1); // if not found
	ii ans; ans.first = lo;
	lo = 0; hi = n - 1; mid = lo;
	while (lo < hi) { // if lower bound is found, find upper bound
		mid = (lo + hi) / 2;
		int res = strncmp(T + SA[mid], P, m);
		if (res > 0) hi = mid; // prune upper half
		else lo = mid + 1; // prune lower half including mid
	} // (notice the selected branch when res == 0)
	if (strncmp(T + SA[hi], P, m) != 0) hi--; // special case
	ans.second = hi;
	return ans;
} // return lower/upperbound as first/second item of the pair, respectively

ii LRS() { // returns a pair (the LRS length and its index)
	int i, idx = 0, maxLCP = -1;
	for (i = 1; i < n; i++) // O(n), start from i = 1
		if (LCP[i] > maxLCP)
			maxLCP = LCP[i], idx = i;
	return ii(maxLCP, idx);
}

int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }


ii LCS() { // returns a pair (the LCS length and its index)
	int i, idx = 0, maxLCP = -1;
	for (i = 1; i < n; i++) // O(n), start from i = 1
		if (owner(SA[i]) != owner(SA[i-1]) && LCP[i] > maxLCP)
			maxLCP = LCP[i], idx = i;
	return ii(maxLCP, idx);
}
```

--- Manacher

```
const char DUMMY = '.';
int manacher(string s) {
	// Add dummy character to not consider odd/even length
	// NOTE: Ensure DUMMY does not appear in input
	// NOTE: Remember to ignore DUMMY when tracing
	int n = s.size() * 2 - 1;
	vector <int> f = vector <int>(n, 0);
	string a = string(n, DUMMY);
	for (int i = 0; i < n; i += 2) a[i] = s[i / 2];

	int l = 0, r = -1, center, res = 0;
	for (int i = 0, j = 0; i < n; i++) {
		j = (i > r ? 0 : min(f[l + r - i], r - i)) + 1;
		while (i - j >= 0 && i + j < n && a[i - j] == a[i + j]) j++;
		f[i] = --j;
		if (i + j > r) {
			r = i + j;
			l = i - j;
		}
		int len = (f[i] + i % 2) / 2 * 2 + 1 - i % 2;
		if (len > res) {
```

```
                res = len;
                center = i;
            }
    } // a[center - f[center]..center + f[center]] is the needed substring
    return res;
}
string longestPalindrome(string s) {
    if(s.size() < 2) return s;
    int max_len = 0;
    int start_idx = 0;
    int i = 0;
    while(i < s.size()) {
        int r_ptr = i;
        int l_ptr = i;
        //find the middle of a palindrome
        while(r_ptr < s.size()-1 && s[r_ptr] == s[r_ptr + 1]) r_ptr++;
        i = r_ptr+1;
        //expand from the middle out
        while(r_ptr < s.size()-1 && l_ptr > 0 && s[r_ptr + 1] == s[l_ptr - 1]) {
            r_ptr++;
            l_ptr--;
        }
        int new_len = r_ptr - l_ptr + 1;
        if(new_len > max_len) {
            start_idx = l_ptr;
            max_len = new_len;
        }
    }
    return s.substr(start_idx, max_len);
}
```

--- Z-Algorithm

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

--- Trie

```
struct vertex {
  char alphabet;
  bool exist;
  vector<vertex*> child;
  vertex(char a): alphabet(a), exist(false) { child.assign(26, NULL); }
};

class Trie {                              // this is TRIE
private:                                  // NOT Suffix Trie
  vertex* root;
public:
  Trie() { root = new vertex('!'); }

  void insert(string word) {              // insert a word into trie
    vertex* cur = root;
    for (int i = 0; i < (int)word.size(); ++i) { // O(n)
      int alphaNum = word[i]-'A';
      if (cur->child[alphaNum] == NULL)   // add new branch if NULL
        cur->child[alphaNum] = new vertex(word[i]);
      cur = cur->child[alphaNum];
    }
    cur->exist = true;
  }
  bool search(string word) {              // true if word in trie
    vertex* cur = root;
    for (int i = 0; i < (int)word.size(); ++i) { // O(m)
      int alphaNum = word[i]-'A';
      if (cur->child[alphaNum] == NULL)   // not found
        return false;
      cur = cur->child[alphaNum];
    }
    return cur->exist;                    // check exist flag
  }

  bool startsWith(string prefix) {        // true if match prefix
    vertex* cur = root;
    for (int i = 0; i < (int)prefix.size(); ++i) {
      int alphaNum = prefix[i]-'A';
      if (cur->child[alphaNum] == NULL)   // not found
        return false;
      cur = cur->child[alphaNum];
    }
    return true;                          // reach here, return true
  }
};
```

--- KMP

```
void kmpPreprocess() {                    // call this first
  int i = 0, j = -1; b[0] = -1;           // starting values
  while (i < m) {                         // pre-process P
    while ((j >= 0) && (P[i] != P[j])) j = b[j]; // different, reset j
    ++i; ++j;                             // same, advance both
    b[i] = j;
  }
}

int kmpSearch() {                         // similar as above
  int freq = 0;
  int i = 0, j = 0;                       // starting values
  while (i < n) {                         // search through T
    while ((j >= 0) && (T[i] != P[j])) j = b[j]; // if different, reset j
    ++i; ++j;                             // if same, advance both
    if (j == m) {                         // a match is found
      ++freq;
      // printf("P is found at index %d in T\n", i-j);
      j = b[j];                           // prepare j for the next
    }
  }
  return freq;
}
```

--- Edit Distance

```
int minDistance(string word1, string word2) {
        int n = word1.size(), m = word2.size();
        int dp[n+1][m+1];

        for(int i=1; i<=n; i++){
            dp[i][0] = i;
        }
        for(int i=1; i<=m; i++){
            dp[0][i] = i;
        }
        dp[0][0] = 0; // no change required when both are empty

        for(int i=1; i<=n; i++){
            for(int j=1; j<=m; j++){
                if (word1[i-1] == word2[j-1]){
                    dp[i][j] = dp[i-1][j-1];
                }else{
                    dp[i][j] = min(min(dp[i-1][j-1],dp[i-1][j]), dp[i][j-1]) + 1;
                }
            }
        }
        return dp[n][m];
    }
```

--- Shortest Palindrome

```
// Given a string, can add characters in front, find min length palindrome
string shortestPalindrome(string s) {
    int n = s.size();
    string rev(s);

    reverse(rev.begin(), rev.end());
    string temp = s + "#" + rev;
    int m = temp.size();

    vector<int> table = build(temp); // kmp build table
    //for(auto& x : table) cout << x << " ";
    return rev.substr(0, n - table[m]) + s;
}
double DEG_to_RAD(double d) { return d*M_PI/180.0; }
double RAD_to_DEG(double r) { return r*180.0/M_PI; }
struct point {
double x, y;                                    // if
need more precision
point() { x = y = 0.0; }                        //
default constructor
point(double _x, double _y) : x(_x), y(_y) {}  //
constructor
bool operator < (point other) const {           //
override < operator
  if (fabs(x-other.x) > EPS)                    // useful
for sorting
    return x < other.x;                         // by
x-coordinate
  return y < other.y;                           // if
tie, by y-coordinate
}
bool operator == (point other) const {// use EPS (1e-9)
when testing equality
 return (fabs(x-other.x) < EPS && (fabs(y-other.y) <
EPS));
}};
double dist(point p1, point p2) {return hypot(p1.x-p2.x,
p1.y-p2.y); }
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
double rad = DEG_to_RAD(theta);                 //
convert to radian
return point(p.x*cos(rad) - p.y*sin(rad),
             p.x*sin(rad) + p.y*cos(rad));
}
struct line { double a, b, c; };                  // most
versatile
// the answer is stored in the third parameter (pass by
reference)
void pointsToLine(point p1, point p2, line &l) {
if (fabs(p1.x-p2.x) < EPS)                      //
vertical line is fine
  l = {1.0, 0.0, -p1.x};                        //
default values
else {
  double a = -(double)(p1.y-p2.y) / (p1.x-p2.x);
  l = {a, 1.0, -(double)(a*p1.x) - p1.y};       // NOTE:
b always 1.0
}
}
struct line2 { double m, c; };                    //
alternative way
int pointsToLine2(point p1, point p2, line2 &l) {
if (p1.x == p2.x) {                             //
vertical line
  l.m = INF;l.c = p1.x;return 0;}
  l.m = (double)(p1.y-p2.y) / (p1.x-p2.x);
  l.c = p1.y - l.m*p1.x;return 1;
}
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){ return areParallel(l1
,l2) && (fabs(l1.c-l2.c) < EPS);}
// returns true (+ intersection point p) if two lines
are intersect
bool areIntersect(line l1, line l2, point &p) {
if (areParallel(l1, l2)) return false;          // no
intersection
// solve system of 2 linear algebraic equations with 2
unknowns
p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);
// special case: test for vertical line to avoid
division by zero
if (fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1.c);
else                  p.y = -(l2.a*p.x + l2.c);
return true;
}
struct vec { double x, y; vec(double _x, double _y) :
x(_x), y(_y) {}};
vec toVec(const point &a, const point &b) {return
vec(b.x-a.x, b.y-a.y);}
vec scale(const vec &v, double s) {return vec(v.x*s,
v.y*s);}
point translate(const point &p, const vec &v) {return
point(p.x+v.x, p.y+v.y);}
// 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) {
// this line is perpendicular to l and pass through p
line perpendicular;
if (fabs(l.b) < EPS) { ans.x = -(l.c);ans.y =
p.y;return;}
if (fabs(l.a) < EPS) {ans.x = p.x;ans.y =
-(l.c);return;}
pointSlopeToLine(p, 1/l.a, perpendicular);      // normal
line
areIntersect(l, perpendicular, ans);}
// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
point b;closestPoint(l, p, b);
vec v = toVec(p, b);ans = translate(translate(p, v), v);
}
// returns the dot product of two vectors a and b
double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }
// returns the squared value of the normalized vector
double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }
double angle(const point &a, const point &o, const point
&b) {
vec oa = toVec(o, a), ob = toVec(o, b);         // a != o
!= b
return acos(dot(oa, ob) / sqrt(norm_sq(oa) *
norm_sq(ob)));}
// 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) {
vec ap = toVec(a, p), ab = toVec(a, b);
double u = dot(ap, ab) / norm_sq(ab);
// formula: c = a + u*ab
c = translate(a, scale(ab, u));                 //
translate a to c
return dist(p, c);                              //
Euclidean distance
}
// returns the distance from p to the line segment ab
defined by
// two points a and b (technically, a has to be
different than 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);} //
closer to a
if (u > 1.0) {c = point(b.x, b.y);return dist(p, b);} //
closer to b
return distToLine(p, a, b, c);                  // use
distToLine
}
// returns the cross product of two vectors a and b
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)) > -EPS;}
// 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 the perimeter of polygon P, which is the sum
of
// Euclidian distances of consecutive line segments
(polygon edges)
double perimeter(const vector<point> &P) {      // by
ref for efficiency
double ans = 0.0;
for (int i = 0; i < (int)P.size()-1; ++i)       // note:
P[n-1] = P[0]
  ans += dist(P[i], P[i+1]);                    // as we
duplicate P[0]
return ans;}
double area(const vector<point> &P) { // returns the
area of polygon P
double ans = 0.0;
for (int i = 0; i < (int)P.size()-1; ++i)       //
Shoelace formula
  ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
return fabs(ans)/2.0;                           // only
do / 2.0 here
}
// 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 n = (int)P.size();
// a point/sz=2 or a line/sz=3 is not convex
if (n <= 3) return false;
bool firstTurn = ccw(P[0], P[1], P[2]);         //
remember one result,
for (int i = 1; i < n-1; ++i)                   // compare
with the others
  if (ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) !=
firstTurn)
    return false;                               //
different -> concave
return true;                                    //
otherwise -> convex
}
// returns 1/0/-1 if point p is inside/on
(vertex/edge)/outside of
// either convex/concave polygon P
int insidePolygon(point pt, const vector<point> &P) {
int n = (int)P.size();
if (n <= 3) return -1;                          // avoid
point or line
bool on_polygon = false;
for (int i = 0; i < n-1; ++i)                   // on
vertex/edge?
```

```
  if (fabs(dist(P[i], pt) + dist(pt, P[i+1]) -
dist(P[i], P[i+1])) < EPS)
    on polygon = true;
if (on polygon) return 0;                        // pt is
on polygon
double sum = 0.0;                                // first
= last point
for (int i = 0; i < n-1; ++i) {
  if (ccw(pt, P[i], P[i+1])) sum += angle(P[i], pt,
P[i+1]); // left turn/ccw
  else sum -= angle(P[i], pt, P[i+1]);            //
right turn/cw
}
return fabs(sum) > M PI ? 1 : -1;                //
360d->in, 0d->out
}
// compute the intersection point between line segment
p-q and line A-B
point lineIntersectSeg(point p, point q, point A, point
B) {
double a = B.y-A.y, b = A.x-B.x, 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 (order matters)
// (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]));
  if (left1 > -EPS) P.push back(Q[i]);           // Q[i]
is on the left
  if (left1*left2 < -EPS)                        //
crosses line AB
    P.push back(lineIntersectSeg(Q[i], Q[i+1], A, B));
}
if (!P.empty() && !(P.back() == P.front()))
  P.push back(P.front());                        // wrap
around
return P;
}
vector<point> CH Andrew(vector<point> &Pts) {    //
overall O(n log n)
int n = Pts.size(), k = 0;
vector<point> H(2*n);
sort(Pts.begin(), Pts.end());                    // sort
the points by x/y
for (int i = 0; i < n; ++i) {                    // build
lower hull
  while ((k >= 2) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
  H[k++] = Pts[i];
}
for (int i = n-2, t = k+1; i >= 0; --i) {
  while ((k >= t) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
  H[k++] = Pts[i];
}
H.resize(k);return H;
}
```

--Bit Operations

```
#define isOn(S, j) (S & (1<<j))
#define setBit(S, j) (S |= (1<<j))
#define clearBit(S, j) (S &= ~(1<<j))
#define toggleBit(S, j) (S ^= (1<<j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1<<n)-1)
#define modulo(S, N) ((S) & (N-1))   // returns S % N,
where N is a power of 2
#define isPowerOfTwo(S) (!(S & (S-1)))
#define nearestPowerOfTwo(S) (1<<lround(log2(S)))
#define turnOffLastBit(S) ((S) & (S-1))
#define turnOnLastZero(S) ((S) | (S+1))
#define turnOffLastConsecutiveBits(S) ((S) & (S+1))
#define turnOnLastConsecutiveZeroes(S) ((S) | (S-1))
```

--Math

```
int mod(int a, int m) {                            //
returns a (mod m)
 return ((a%m) + m) % m;                          //
ensure positive answer
}

int modPow(int b, int p, int m) {                  //
assume 0 <= b < m
 if (p == 0) return 1;
 int ans = modPow(b, p/2, m);                     // this
is O(log p)
 ans = mod(ans*ans, m);                           //
double it first
 if (p&1) ans = mod(ans*b, m);                    // *b if
p is odd
 return ans;                                      // ans
always in [0..m-1]
}

int extEuclid(int a, int b, int &x, int &y) {    // pass
x and y by ref
 int xx = y = 0;
 int yy = x = 1;
 while (b) {                                     //
repeats until b == 0
   int q = a/b;
   tie(a, b) = tuple(b, a%b);
   tie(x, xx) = tuple(xx, x-q*xx);
   tie(y, yy) = tuple(yy, y-q*yy);
 }
 return a;                                       //
returns gcd(a, b)
}

int modInverse(int b, int m) {                    //
returns b^(-1) (mod m)
 int x, y;
 int d = extEuclid(b, m, x, y);                  // to
get b*x + m*y == d
 if (d != 1) return -1;                          // to
indicate failure
 // b*x + m*y == 1, now apply (mod m) to get b*x == 1
(mod m)
 return mod(x, m);
}

ll _sieve_size;
bitset<10000010> bs;                              // 10^7
is the rough limit
vll p;                                            //
compact list of primes

void sieve(ll upperbound) {                       //
range = [0..upperbound]
 _sieve_size = upperbound+1;                     // to
include upperbound
 bs.set();                                       // all
1s
 bs[0] = bs[1] = 0;                              //
except index 0+1
 for (ll i = 2; i < _sieve_size; ++i) if (bs[i]) {
   // cross out multiples of i starting from i*i
   for (ll j = i*i; j < _sieve_size; j += i) bs[j] = 0;
   p.push_back(i);                               // add
prime i to the list
 }
}

bool isPrime(ll N) {                              // good
enough prime test
 if (N < _sieve_size) return bs[N];              // O(1)
for small primes
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N;
++i)
   if (N%p[i] == 0)
     return false;
 return true;                                    // slow
if N = large prime
} // note: only guaranteed to work for N <= (last prime
in vll p)^2

// second part

vll primeFactors(ll N) {                          //
pre-condition, N >= 1
 vll factors;
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N;
++i)
   while (N%p[i] == 0) {                         // found
a prime for N
     N /= p[i];                                  //
remove it from N
     factors.push_back(p[i]);
   }
 if (N != 1) factors.push_back(N);               //
remaining N is a prime
 return factors;
}

// third part

int numPF(ll N) {
```

```
 int ans = 0;
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N;
++i)
   while (N%p[i] == 0) { N /= p[i]; ++ans; }
 return ans + (N != 1);
}

int numDiffPF(ll N) {
 int ans = 0;
 for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i) {
   if (N%p[i] == 0) ++ans;                      // count
this prime factor
   while (N%p[i] == 0) N /= p[i];               // only
once
 }
 if (N != 1) ++ans;
 return ans;
}

ll sumPF(ll N) {
 ll ans = 0;
 for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i)
   while (N%p[i] == 0) { N /= p[i]; ans += p[i]; }
 if (N != 1) ans += N;
 return ans;
}

int numDiv(ll N) {
 int ans = 1;                                   // start
from ans = 1
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N;
++i) {
   int power = 0;                               // count
the power
   while (N%p[i] == 0) { N /= p[i]; ++power; }
   ans *= power+1;                              //
follow the formula
 }
 return (N != 1) ? 2*ans : ans;                 // last
factor = N^1
}

ll sumDiv(ll N) {
 ll ans = 1;                                    // start
from ans = 1
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N;
++i) {
   ll multiplier = p[i], total = 1;
   while (N%p[i] == 0) {
     N /= p[i];
     total += multiplier;
     multiplier *= p[i];
   }                                            // total
for
   ans *= total;                                // this
prime factor
 }
 if (N != 1) ans *= (N+1);                      //
N^2-1/N-1 = N+1
 return ans;
}

ll EulerPhi(ll N) {
 ll ans = N;                                    // start
from ans = N
 for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N;
++i) {
   if (N%p[i] == 0) ans -= ans/p[i];            // count
unique
   while (N%p[i] == 0) N /= p[i];               // prime
factor
 }
 if (N != 1) ans -= ans/N;                      // last
factor
 return ans;
}

--zs Matrix code
const int mod = int(1e9) + 7;
const int DIM = 52;

struct Matrix {
 int a[DIM][DIM];
 int *operator [] (int r) { return a[r]; };

 Matrix(int x = 0) {
   memset(a, 0, sizeof a);
   if (x)
   {
       for(int i = 0; i < DIM; i++) a[i][i] = x;
   }
 }
} const I(1);

Matrix operator * (Matrix A, Matrix B) {
 const ll mod2 = ll(mod) * mod;
 Matrix C;
for(int i = 0; i < DIM; i++)
{
    for(int j = 0; j < DIM; j++)
    {
       ll w = 0;
       for(int k = 0; k < DIM; k++)
       {
           w += ll(A[i][k]) * B[k][j];
           if (w >= mod2) w -= mod2;
       }
   C[i][j] = w % mod;
   }
 }
 return C;
}

Matrix operator ^ (Matrix A, ll b) {
 Matrix R = I;
 for (; b > 0; b /= 2) {
   if (b % 2) R = R*A;
   A = A*A;
 }
 return R;
}

-- - FFT
typedef complex < double > Base;
int rev[MN];
Base wlen_pw[MN];
void eval(Base a[], int n, bool invert) {
 for (int i = 0; i < n; ++i)
   if (i < rev[i]) swap(a[i], a[rev[i]]);
 for (int len = 2; len <= n; len <<= 1) {
   double ang = 2 * M_PI / len * (invert ? -1 : +1);
   int len2 = len >> 1;
   Base wlen(cos(ang), sin(ang));
   wlen_pw[0] = Base(1, 0);
   for (int i = 1; i < len2; ++i)
     wlen_pw[i] = wlen_pw[i - 1] * wlen;
   for (int i = 0; i < n; i += len) {
     Base t,
     * pu = a + i,
       * pv = a + i + len2,
       * pu_end = a + i + len2,
       * pw = wlen_pw;
     for (; pu != pu_end; ++pu, ++pv, ++pw) {
       t = * pv * * pw;
       * pv = * pu - t;
       * pu += t;
     }
   }
 }
 if (invert)
   for (int i = 0; i < n; ++i)
     a[i] /= n;
}
void calc_rev(int n, int log_n) {
 for (int i = 0; i < n; ++i) {
   rev[i] = 0;
   for (int j = 0; j < log_n; ++j)
     if (i & (1 << j))
       rev[i] |= 1 << (log_n - 1 - j);
 }
}
// multiply a[0] * a[1] and store into a[2]
void multiply(Base a[][MN], int n) {
 int outN = 1, lg = 1;
 while (outN < n) outN <<= 1, ++lg;
 outN <<= 1;
 calc_rev(outN, lg);
 eval(a[0], outN, false);
 eval(a[1], outN, false); -
 47 -
   for (int i = 0; i < outN; ++i)
     a[2][i] = a[0][i] * a[1][i];
 eval(a[2], outN, true);
}

-- - Gaussian Elimination#include <cmath>

#include <cstdio>

using namespace std;
#define MAX_N 3
// adjust this value as needed
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;
   // the forward elimination phase
   -
   45 -
     for (j = i + 1; j < N; j++)
       // which row has largest column value
       if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i]))
         l = j;
   // remember this row l
   // swap this pivot row, reason: minimize floating
point error
   for (k = i; k <= N; k++)
     // t is a temporary double variable
     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++)
     // the actual forward elimination phase
     for (k = N; k >= i; k--)
       Aug.mat[j][k] -= Aug.mat[i][k] * Aug.mat[j][i] /
Aug.mat[i][i];
 }
 ColumnVector Ans;
 // the back substitution phase
 for (j = N - 1; j >= 0; j--) {
   // start from back
   for (t = 0.0, k = j + 1; k < N; k++) t +=
Aug.mat[j][k] * Ans.vec[k];
   Ans.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j]; //
the answer is here
 }
 return Ans;
}
int main() {
   AugmentedMatrix
   Aug.mat[0][0] =
     Aug.mat[1][0] =
     Aug.mat[2][0] =
     Aug;
   1;
   Aug.mat[0][1] = 1;
   Aug.mat[0][2] = 2;
   Aug.mat[0][3] = 9;
   2;
   Aug.mat[1][1] = 4;
   Aug.mat[1][2] = -3;
   Aug.mat[1][3] = 1;
   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;
 }
 -- - Pollards rho big integer factoring
import java.util.*;
class Pollardsrho {
 public static long mulmod(long a, long b, long c) { //
returns (a * b) % c, and minimize
   overflow
   long x = 0, y = a % c;
   while (b > 0) {
     if (b % 2 == 1) x = (x + y) % c;
     y = (y * 2) % c;
     b /= 2;
   }
   return x % c;
 }
 public static long abs_val(long a) {
   return a >= 0 ? a : -a;
 }
 public static long gcd(long a, long b) {
   return b == 0 ? a : gcd(b, a % b);
 } // standard
 gcd
 public static long pollard_rho(long n) {
   int i = 0, k = 2;
   long x = 3, y = 3;
   // random seed = 3, other values possible
   while (true) {
     i++;
     x = (mulmod(x, x, n) + n - 1) % n;
     // generating function
     long d = gcd(abs_val(y - x), n);
     // the key insight
     if (d != 1 && d != n) return d;
     // found one non-trivial factor
     if (i == k) {
       y = x;
       k *= 2;
     }
   } -
   46 -
 }
 public static void main(String[] args) {
   long n = 2063512844981574047 L; // we assume that n
is not a large prime
   long ans = pollard_rho(n);
   // break n into two non trivial factors
   if (ans > n / ans) ans = n / ans;
   // make ans the smaller factor
   System.out.println(ans + " " + (n / ans)); // should
be: 1112041493 1855607779
 }
}--;
```

```
--Python Miller rabin
import random
# Rabin_Miller primality check. Tests whether or not a
number
# is prime with a failure rate of: (1/2)^certainty
def isPrime(n, certainty = 12 ):
   if(n < 2): return False
   if(n != 2 and (n & 1) == 0): return False
   s = n-1
   while((s & 1) == 0): s >>= 1
   for _ in range(certainty):
       r = random.randrange(n-1) + 1
       tmp = s
       mod = pow(r,tmp,n)
       while(tmp != n-1 and mod != 1 and mod != n-1):
           mod = (mod*mod) % n
           tmp <<= 1
       if (mod != n-1 and  (tmp & 1) == 0): return False
   return True
```

```
--Newton Horner root finding pseudocode lol
Xn+1 = Xn - p(x)/p'(x)
p'(x) either use power rule or rufini p'(x)=q(x)
Synthetic division to 'deflate' the equation
Repeat till all roots

--Rational root theorem says that all rational roots
are+-{factorsOf(biggestCoeff)/factorsOf(smallestcoeff)}
Just try em all hehe
```

Stupid tips:
1.If u get WA, try putting a constraint on the answer.
2. Make sure nextInt's are ints and nextLongs are longs lmao
3. Sorting queries is a legitimate strategy. So like parallel process the answers. Amazing
4. Dont fuck with offset if u don't have have to. That extra null element at the start will not kill u. But being wrong because of a 1 offset will.
5. Memoizing can sometimes cost more than just redoing the thing.
6. Make sure default values aren't out of problem specification range.
7. Dont like typing static all the time? Don't code in the main class!
8. Be very very careful with limits of a long and limits of an int
9. Remember to finish taking in all the input! Dont break and leave stuff hanging.
10. Be careful of bitshifting longs -> e.g. ((long)1 <<50)

Basic skeleton of DP
1.Define subproblems
2. Guess
- what are we optimising in our guess?
how many guesses are there?
3. Relation
How do subproblems relate to each other?
How does previous encoding help current guesses?
Is there not enough information in the previous subproblem?
4. Time
Time = Subproblems*Guesses
Is asymptotic complexity good enough to pass in time?
5. Original problem
Did the DP solve the original problem?
Which encoded state stores what we want?
6. DAG
Memoized Recursive tree (Top-down) or Topological Order(Bottom-up)
Tabular visualization
Did we need so many states?

```
Python input speedup
inputs = sys.stdin.read().splitlines()

Inner_product,adjacent_pairs,partial_sum,adjacent_difference,transform,for_each,unique,swap,move,generate

search,find_first_of,find_end,equal_range,lower/upper_bound,mismatch,count_if,find_if,all_of,any_of,none_of

set_difference,set_union,includes,set_intersection,symmetric_difference
nth_element does O(n) order stat
```

```
--Fenwick Tree

#define LSOne(S) ((S) & -(S))                    // the
key operation

typedef long long ll;                            // for
extra flexibility
typedef vector<ll> vll;
typedef vector<int> vi;

class FenwickTree {                              //
index 0 is not used
private:
 vll ft;                                         //
internal FT is an array
public:
 FenwickTree(int m) { ft.assign(m+1, 0); }       //
create an empty FT

 void build(const vll &f) {
   int m = (int)f.size()-1;                      // note
f[0] is always 0
   ft.assign(m+1, 0);
   for (int i = 1; i <= m; ++i) {                // O(m)
     ft[i] += f[i];                              // add
this value
     if (i+LSOne(i) <= m)                        // i has
parent
       ft[i+LSOne(i)] += ft[i];                  // add
to that parent
   }
 }

 FenwickTree(const vll &f) { build(f); }         //
create FT based on f

 FenwickTree(int m, const vi &s) {               //
create FT based on s
   vll f(m+1, 0);
   for (int i = 0; i < (int)s.size(); ++i)       // do
the conversion first
     ++f[s[i]];                                  // in
O(n)
   build(f);                                     // in
O(m)
 }

 ll rsq(int j) {                                 //
returns RSQ(1, j)
   ll sum = 0;
   for (; j; j -= LSOne(j))
     sum += ft[j];
   return sum;
 }

 ll rsq(int i, int j) { return rsq(j) - rsq(i-1); } //
inc/exclusion

 // updates value of the i-th element by v (v can be
+ve/inc or -ve/dec)
 void update(int i, ll v) {
   for (; i < (int)ft.size(); i += LSOne(i))
     ft[i] += v;
 }

 int select(ll k) {                              // O(log
m)
   int p = 1;
   while (p*2 < (int)ft.size()) p *= 2;
   int i = 0;
   while (p) {
     if (k > ft[i+p]) {
       k -= ft[i+p];
       i += p;
     }
     p /= 2;
   }
   return i+1;
 }
};

class RUPQ {                                     // RUPQ
variant
private:
 FenwickTree ft;                                 //
internally use PURQ FT
public:
 RUPQ(int m) : ft(FenwickTree(m)) {}
 void range_update(int ui, int uj, int v) {
   ft.update(ui, v);                             // [ui,
ui+1, .., m] +v
   ft.update(uj+1, -v);                          //
[uj+1, uj+2, .., m] -v
 }                                               // [ui,
ui+1, .., uj] +v
 ll point_query(int i) { return ft.rsq(i); }     //
rsq(i) is sufficient
};

class RURQ  {                                    // RURQ
variant
private:                                         //
needs two helper FTs
 RUPQ rupq;                                      // one
RUPQ and
 FenwickTree purq;                               // one
PURQ
public:
 RURQ(int m) : rupq(RUPQ(m)), purq(FenwickTree(m)) {} //
initialization
 void range_update(int ui, int uj, int v) {
   rupq.range_update(ui, uj, v);                 // [ui,
ui+1, .., uj] +v
   purq.update(ui, v*(ui-1));                    //
-(ui-1)*v before ui
   purq.update(uj+1, -v*uj);                     //
+(uj-ui+1)*v after uj
 }
 ll rsq(int j) {
   return rupq.point_query(j)*j -                //
optimistic calculation
          purq.rsq(j);                           //
cancelation factor
 }
 ll rsq(int i, int j) { return rsq(j) - rsq(i-1); } //
standard
};

int main() {
 vll f = {0,0,1,0,1,2,3,2,1,1,0};                // index
0 is always 0
 FenwickTree ft(f);
 printf("%lld\n", ft.rsq(1, 6)); // 7 => ft[6]+ft[4] =
5+2 = 7
 printf("%d\n", ft.select(7)); // index 6, rsq(1, 6) ==
7, which is >= 7
 ft.update(5, 1); // update demo
 printf("%lld\n", ft.rsq(1, 10)); // now 12
 printf("=====\n");
 RUPQ rupq(10);
 RURQ rurq(10);
 rupq.range_update(2, 9, 7); // indices in [2, 3, .., 9]
updated by +7
 rurq.range_update(2, 9, 7); // same as rupq above
 rupq.range_update(6, 7, 3); // indices 6&7 are further
updated by +3 (10)
 rurq.range_update(6, 7, 3); // same as rupq above
 // idx = 0 (unused) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
|10
 // val = -          | 0 | 7 | 7 | 7 | 7 |10 |10 | 7 | 7
| 0
 for (int i = 1; i <= 10; i++)
   printf("%d -> %lld\n", i, rupq.point_query(i));
 printf("RSQ(1, 10) = %lld\n", rurq.rsq(1, 10)); // 62
 printf("RSQ(6, 7) = %lld\n", rurq.rsq(6, 7)); // 20
 return 0;
}
```

--- DSU

```
struct DSU2{ //with rollback
	vi par;
	int cc;
	vector<ii>updates;
	void init(int n){
		par.resize(n+1);
		for(int i=0; i<=n; i++)par[i]=i;
		cc=n;
	}
	int rt(int u){
		if(par[u]!=u)return rt(par[u]);
		return par[u];
	}
	bool merge(int u, int v){
		u=rt(u); v=rt(v);
		if(u==v){
			updates.eb(-1,-1);
			return 0;
		}
		if(rand()%2)swap(u,v);
		updates.eb(v,par[v]);
		par[v]=u;
		cc--;
		return 1;
```

```
	}
	bool sameset(int u, int v){
		u=rt(u); v=rt(v);
		return u==v;
	}
	void rollback(){
		if(updates.empty())return;
		int v = updates.back().fi;
		int pv = updates.back().se;
		int curp = par[v]; //aka u
		//do whatever you want
		par[v]=pv;
	}
};
```

---Centroid Decomposition

```
const int mxn = 1e5;
int siz[mxn];
bool vis[mxn];
void dfs_sz(int u, int p){
	siz[u]=1;
	for(int v:adj[u]){
		if(v==p||vis[v])continue;
		dfs_sz(v,u);
		siz[u]+=siz[v];
	}
}
int centroid(int u, int p, int sizz){
	for(int v:adj[u]){
		if(v==p||vis[v])continue;
		if(siz[v]*2>sizz)return
centroid(v,u,sizz);
	}
	return u;
}
void solve(int u=0, int p=-1){
	dfs_sz(u,-1);
	int ct = centroid(u,-1,siz[u]);
	//do whatever you want to solve with
your centroid, prolly dfs ct
	/* in func dfs
	 * for(int v:adj[ct]){
	 *		if(v==p||vis[v])continue;
	 *
	 * }
	 *
	vis[ct]=1;
	for(int v:adj[u]){
		if(v==p||vis[v])continue;
		solve(v,u);
	}
}
```

--- Combinatorics

```
const int MOD = (int)1e9+7;
const int mxn = 2e5+5;
int fact[mxn],ifact[mxn];
void multi(ll &a, ll b){
	a%=MOD; b%=MOD; a*=b; a%=MOD;
}

void add(ll &a, ll b){
	a%=MOD; b%=MOD; a+=b; a%=MOD;
}

ll modpow(ll a, ll b){
	ll res = 1ll;
	while(b){
		if(b&1)multi(res,a);
		multi(a,a);
		b>>=1;
	}
	return res;
}

void init(){
	fact[0]=ifact[0]=1ll;
	for(int i=1; i<mxn; i++){
		fact[i]=fact[i-1]*1ll*i%MOD;
		ifact[i]=modpow(fact[i],MOD-2);
	}
}

ll nCr(ll n, ll r){
	ll res = fact[n];
	multi(res,ifact[r]);
	multi(res,ifact[n-r]);
	return res;
}
```

---Convex Hull (Graham's Scan)

```
struct pt {
    double x, y;
};

bool cmp(pt a, pt b) {
    return a.x < b.x || (a.x == b.x && a.y <
b.y);
}

bool cw(pt a, pt b, pt c) {
    return
a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
}

bool ccw(pt a, pt b, pt c) {
    return
a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;
}

void convex_hull(vector<pt>& a) {
    if (a.size() == 1)
        return;
```

```
    sort(a.begin(), a.end(), &cmp);
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i],
p2)) {
            while (up.size() >= 2 &&
!cw(up[up.size()-2], up[up.size()-1], a[i]))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i],
p2)) {
            while(down.size() >= 2 &&
!ccw(down[down.size()-2], down[down.size()-1],
a[i]))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

---Euler's totient function

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    phi[0] = 0;
    phi[1] = 1;
    for (int i = 2; 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;
        }
```

```
        }
}
ll lcm(ll a, ll b) {
    return (a / gcd(a, b))*b;
}
__builtin_popcount (4) will return 1
```

-Minimax

```
def minimax(gamestate, depth,is_maxplayer,alpha,beta):
    if depth==0 or gameover:
        return eval(gamestate)
    if(maxplayer):
        maxEval = -INF
        for child in children(gamestate):
            eval = minimax(child,
depth-1,false,alpha,beta)
            maxEval = max(maxEval,eval)
            alpha = max(alpha,eval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = INF
        for child in children(gamestate):
            eval =
minimax(child,depth-1,true,alpha,beta)
            minEval = min(minEval,eval)
            beta = min(beta,eval)
            if beta <= alpha:
                break
        return minEval
```

--Binary eval

```
public class Binary_evalutation{
  public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      int n = in.nextInt(); in.nextLine();
      for (int i = 0; i < n; i++) {

if(eval("("+in.nextLine()+")")){System.out.println("0");}e
lse{System.out.println("1");}
      }
      in.close();
  }
  public static boolean eval(String E){
      Queue<Character> post = in2post(E);
      Stack<Boolean> st1= new Stack<>();
      Stack<Boolean> st2= new Stack<>();
      while(!post.isEmpty()){
          switch(post.poll()){
              case '1':
st1.push(true);st2.push(true);break;
              case '0':
st1.push(false);st2.push(false);break;
              case 'x':
st1.push(true);st2.push(false);break;
              case 'X':
st1.push(false);st2.push(true);break;
              case '|':
st1.push(st1.pop()|st1.pop());st2.push(st2.pop()|st2.pop()
);break;
              case '&':
st1.push(st1.pop()&st1.pop());st2.push(st2.pop()&st2.pop()
);break;
              case '^':
st1.push(st1.pop()^st1.pop());st2.push(st2.pop()^st2.pop()
);break;
          }
      }
      if(st1.pop()==st2.pop())return true;
      return false;
  }
  public static Queue<Character> in2post(String E){
      Queue<Character> post = new LinkedList<>();
      Stack<Character> buff = new Stack<>();
      for (int i = 0; i < E.length(); i++) {
          char c = E.charAt(i);
          if(c=='x'||c=='X'||c=='1'||c=='0')post.add(c);
          else if(c=='|'||c=='&'||c=='^')buff.push(c);
          else if(c=='(')buff.push('(');
          else if(c==')'){
              while((c=buff.pop())!='(')post.add(c);
          }
      }
      return post;
  }
}
```

--LRS

```
public class LongestRepeatedSubstring {
   public static void main(String[] args) {
     String str = "ABC$BCA$CAB";SuffixArray sa = new
SuffixArray(str);
     System.out.printf("LRS(s) of %s is/are: %s\n", str,
sa.lrs());}
   public static class SuffixArray {
     int ALPHABET_SZ = 256, N;int[] T, lcp, sa, sa2, rank, tmp, c;
     public SuffixArray(String str) {this(toIntArray(str));}
```

```
    private static int[] toIntArray(String s) {int[] text = new
int[s.length()];
        for (int i = 0; i < s.length(); i++) text[i] = s.charAt(i);
return text;}
    public SuffixArray(int[] text) {
        T = text;N = text.length;sa = new int[N];sa2 = new
int[N];rank = new int[N];
        c = new int[Math.max(ALPHABET_SZ, N)];construct();kasai();}
    private void construct() {int i, p, r;
      for (i = 0; i < N; ++i) c[rank[i] = T[i]]++;
      for (i = 1; i < ALPHABET_SZ; ++i) c[i] += c[i - 1];
      for (i = N - 1; i >= 0; --i) sa[--c[T[i]]] = i;
      for (p = 1; p < N; p <<= 1) {
        for (r = 0, i = N - p; i < N; ++i) sa2[r++] = i;
        for (i = 0; i < N; ++i) if (sa[i] >= p) sa2[r++] = sa[i]
- p;
        Arrays.fill(c, 0, ALPHABET_SZ, 0);
        for (i = 0; i < N; ++i) c[rank[i]]++;
        for (i = 1; i < ALPHABET_SZ; ++i) c[i] += c[i - 1];
        for (i = N - 1; i >= 0; --i) sa[--c[rank[sa2[i]]]] =
sa2[i];
        for (sa2[sa[0]] = r = 0, i = 1; i < N; ++i) {
          if (!(rank[sa[i - 1]] == rank[sa[i]]
              && sa[i - 1] + p < N
              && sa[i] + p < N
              && rank[sa[i - 1] + p] == rank[sa[i] + p])) r++;
          sa2[sa[i]] = r;
        }tmp = rank;rank = sa2;sa2 = tmp;if (r == N - 1)
break;ALPHABET_SZ = r + 1;}}
    private void kasai() {lcp = new int[N];int[] inv = new
int[N];
      for (int i = 0; i < N; i++) inv[sa[i]] = i;
      for (int i = 0, len = 0; i < N; i++) {
        if (inv[i] > 0) {int k = sa[inv[i] - 1];
          while ((i + len < N) && (k + len < N) && T[i + len] ==
T[k + len]) len++;
          lcp[inv[i] - 1] = len;if (len > 0) len--;}}}
     // Finds the LRS(s) (Longest Repeated Substring) that occurs
in a string.
    // Traditionally we are only interested in substrings that
appear at
    // least twice, so this method returns an empty set if this
is not the case.
    // @return an ordered set of longest repeated substrings
    public TreeSet<String> lrs() {
      int max_len = 0;TreeSet<String> lrss = new TreeSet<>();
      for (int i = 0; i < N; i++) {
        if (lcp[i] > 0 && lcp[i] >= max_len) {
if (lcp[i] > max_len) lrss.clear();    max_len =
lcp[i];lrss.add(new String(T, sa[i], max_len));}}return lrss;}
    public void display()
{System.out.printf("-----i-----SA-----LCP---Suffix\n");
      for (int i = 0; i < N; i++) {int suffixLen = N -
sa[i];String suffix = new String(T, sa[i], suffixLen);
        System.out.printf("% 7d % 7d % 7d %s\n", i, sa[i],
lcp[i], suffix);}}}}
```

```
--Circles
import math
def DEG_to_RAD(d): return d*math.pi/180.0
def RAD_to_DEG(r): return r*180.0/math.pi
class point:
   def __init__(self, _x, _y):                      # int
or double
       self.x = _x
       self.y = _y
   def getX(self):
       return self.x
   def getY(self):
       return self.y
# returns 0/1/2 for inside/border/outside, respectively
def insideCircle(p, c, r):                            # all
integer version
   dx = p.getX()-c.getX()
   dy = p.getY()-c.getY()
   Euc = dx*dx + dy*dy
   rSq = r*r
   return 1 if Euc < rSq else (0 if Euc == rSq else -1)
def circle2PtsRad(p1, p2, r, c_list):
   # to get the other center, reverse p1 and p2
   d2 = (p1.getX()-p2.getX()) * (p1.getX()-p2.getX()) +
(p1.getY()-p2.getY()) * (p1.getY()-p2.getY())
   det = r*r / d2 - 0.25
   if det < 0.0: return False
   h = math.sqrt(det)
   c_list[0].x = (p1.getX()+p2.getX()) * 0.5 +
(p1.getY()-p2.getY()) * h
   c_list[0].y = (p1.getY()+p2.getY()) * 0.5 +
(p2.getX()-p1.getX()) * h
   return True
def main():
   # circle equation, inside, border, outside
   pt = point(2, 2)
   r = 7
   inside = point(8, 2)
   print(insideCircle(inside, pt, r))              # 1,
inside
   border = point (9, 2)
   print(insideCircle(border, pt, r))              # 0,
at border
   outside = point(10, 2)
   print(insideCircle(outside, pt, r))             # -1,
outside

   d = 2*r
   print("Diameter = ", "{:.2f}".format(d))
   c = math.pi*d
   print("Circumference (Perimeter) = ",
"{:.2f}".format(c))
   A = math.pi*r*r
   print("Area of circle = ", "{:.2f}".format(A))

   print("Length of arc   (central angle = 60 degrees)
= ", "{:.2f}".format(60.0/360.0 * c))
   print("Length of chord (central angle = 60 degrees)
= ", "{:.2f}".format(math.sqrt((2*r*r) * (1 -
math.cos(DEG_to_RAD(60.0))))))
   print("Area of sector  (central angle = 60 degrees)
= ", "{:.2f}".format(60.0/360.0 * A))

   p1 = point(0, 0)
   p2 = point(0.0, -1.0)
   ans = [point(0, 0)]                             # use
a wrapper
   circle2PtsRad(p1, p2, 2.0, ans)
   print("One of the center is (",
"{:.2f}".format(ans[0].getX()), ",",
"{:.2f}".format(ans[0].getY()), ")")
   circle2PtsRad(p2, p1, 2.0, ans)                 #
reverse p1 with p2
   print("The other center is (",
"{:.2f}".format(ans[0].getX()), ",",
"{:.2f}".format(ans[0].getY()), ")")
main()
```

```
--Delauney triangulation
// Does not handle
// degenerate cases (from O'Rourke, Computational
Geometry in C)
// Running time: O(n^4)
// INPUT:    x[] = x-coordinates
//           y[] = y-coordinates
// OUTPUT:   triples = a vector containing m triples of
indices
corresponding to triangle vertices

using namespace std;
typedef double T;
struct triple {
   int i, j, k;
   triple() {}
   triple(int i, int j, int k) : i(i), j(j), k(k) {}
};
vector<triple> delaunayTriangulation(vector<T>& x,
vector<T>& y) {
   int n = x.size();
   vector<T> z(n);
   vector<triple> ret;

   for (int i = 0; i < n; i++)
       z[i] = x[i] * x[i] + y[i] * y[i];

   for (int i = 0; i < n-2; i++) {
       for (int j = i+1; j < n; j++) {
       for (int k = i+1; k < n; k++) {
```

```
          if (j == k) continue;
          double xn = (y[j]-y[i])*(z[k]-z[i]) -
(y[k]-y[i])*(z[j]-z[i]);
          double yn = (x[k]-x[i])*(z[j]-z[i]) -
(x[j]-x[i])*(z[k]-z[i]);
          double zn = (x[j]-x[i])*(y[k]-y[i]) -
(x[k]-x[i])*(y[j]-y[i]);
          bool flag = zn < 0;
          for (int m = 0; flag && m < n; m++)
          flag = flag && ((x[m]-x[i])*xn +
                  (y[m]-y[i])*yn +
                  (z[m]-z[i])*zn <= 0);
          if (flag) ret.push_back(triple(i, j, k));
        }
        }
    }
    return ret;
}
int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);
    //expected: 0 1 3
    //          0 3 2
    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j,
tri[i].k);
    return 0;
}
```

--FastExp

```
using namespace std;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
T power(T x, int k) {
 T ret = 1;
  while(k) {
   if(k & 1) ret *= x;
   k >>= 1; x *= x;
 }
 return ret;
}
VVT multiply(VVT& A, VVT& B) {
 int n = A.size(), m = A[0].size(), k = B[0].size();
 VVT C(n, VT(k, 0));
  for(int i = 0; i < n; i++)
   for(int j = 0; j < k; j++)
     for(int l = 0; l < m; l++)
       C[i][j] += A[i][l] * B[l][j];

 return C;
}
VVT power(VVT& A, int k) {
 int n = A.size();
 VVT ret(n, VT(n)), B = A;
 for(int i = 0; i < n; i++) ret[i][i]=1;

 while(k) {
   if(k & 1) ret = multiply(ret, B);
   k >>= 1; B = multiply(B, B);
 }
 return ret;
}
int main()
{
 /* Expected Output:
    2.37^48 = 9.72569e+17
    376 264 285 220 265
    550 376 529 285 484
    484 265 376 264 285
    285 220 265 156 264
    529 285 484 265 376 */
 double n = 2.37;
 int k = 48;
 cout << n << "^" << k << " = " << power(n, k) << endl;
 double At[5][5] = {
   { 0, 0, 1, 0, 0 },
   { 1, 0, 0, 1, 0 },
   { 0, 0, 0, 0, 1 },
   { 1, 0, 0, 0, 0 },
   { 0, 1, 0, 0, 0 } };
 vector <vector <double> > A(5, vector <double>(5));
 for(int i = 0; i < 5; i++)
   for(int j = 0; j < 5; j++)
     A[i][j] = At[i][j];
 vector <vector <double> > Ap = power(A, k);
 cout << endl;
 for(int i = 0; i < 5; i++) {
   for(int j = 0; j < 5; j++)
     cout << Ap[i][j] << " ";
   cout << endl;
 }
}
```

--3d Geo

```
public class Geom3D {
 // distance from point (x, y, z) to plane aX + bY + cZ
+ d = 0
 public static double ptPlaneDist(double x, double y,
double z,
     double a, double b, double c, double d) {
   return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a
+ b*b + c*c);
 }
 // distance between parallel planes aX + bY + cZ + d1
= 0 and
 // aX + bY + cZ + d2 = 0
 public static double planePlaneDist(double a, double
b, double c,
     double d1, double d2) {
   return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b +
c*c);
 }
 public static final int LINE = 0;
 public static final int SEGMENT = 1;
 public static final int RAY = 2;
 public static double ptLineDistSq(double x1, double
y1, double z1,
     double x2, double y2, double z2, double px, double
py, double pz,
     int type) {
   double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) +
(z1-z2)*(z1-z2);
   double x, y, z;
   if (pd2 == 0) {x = x1;y = y1;z = z1;} else {
     double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) +
(pz-z1)*(z2-z1)) / pd2;
     x = x1 + u * (x2 - x1);
     y = y1 + u * (y2 - y1);
     z = z1 + u * (z2 - z1);
     if (type != LINE && u < 0) {x = x1;y = y1;z = z1;}
     if (type == SEGMENT && u > 1.0) {x = x2;y = y2;z =
z2;}}
   return (x-px)*(x-px) + (y-py)*(y-py) +
(z-pz)*(z-pz);}
 public static double ptLineDist(double x1, double y1,
double z1,
     double x2, double y2, double z2, double px, double
py, double pz,
     int type) {
   return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2,
z2, px, py, pz, type));
 }}
```