ButterRoti ICPC Team Notebook (2017-18)

Contents

1 Misc

1.1 Build

```
{
"cmd": ["g++ -std=c++14 -g -Wall '${file}' && timeout 15s '${file_path}/./a.out'<'${file_path}/input.txt'>'${file_path}/output.txt'"],
"shell":true
}
```

1.2 Snippet

```cpp
#include <bits/stdc++.h>

using namespace std;

template<typename T> using V = vector<T>;
template<typename T, typename V> using P = pair<T, V>;
template<typename T> using min_heap = priority_queue<T, V<T>, greater<T>>;

using LL = long long;
using ll = LL;
using LD = long double;
using ld = long double;

#define fi first
#define ff first
#define se second
#define ss second
#define pp push_back
#define pb pp
#define endl '\n'
#define SYNC std::ios::sync_with_stdio(false); cin.tie(NULL);
#define ALL(v) v.begin(), v.end()
#define FOR0(i,n) for(int i=0, _##i=(n); i<_##i; ++i)
#define FOR(i,l,r) for(int i=(l), _##i=(r); i<_##i; ++i)
#define FORD(i,l,r) for(int i=(r), _##i=(l); --i>=_##i; )
#define rep(i,a) FOR0(i, a)
#define repn(i,a) FOR(i, 1, a + 1)
#define REP(i, n) rep(i, n)
#define REPN(i, n) repn(i, n)
#define SZ(a) ((int)((a).size()))
#define mp make_pair
#define dzx cerr << "here";
#define her cerr << "HERE "
#define pii pair<int,int>
#define ii pii
#define en(v) *(--v.end())

const int MOD = (int)1e9 + 7, inf = 0x3f3f3f3f;
const ll INF = 0x3f3f3f3f3f3f3f3f;

int32_t main() {SYNC;

    return 0;
```

```
}
```

1.3 Stack Size Increase

```
#include <sys/resource.h>

int main(){
  rlimit R;
  getrlimit(RLIMIT_STACK, &R);
  R.rlim_cur = R.rlim_max;
  setrlimit(RLIMIT_STACK, &R);
}
```

1.4 Variadic Multiplication and Addition

```

const int MOD = (int)1e9 + 7;

int add(){ return 0; }

template<typename... T> int add(int a, T... arg){
  int b = add(arg...);
  return (a + b >= MOD ? a + b - MOD : a + b);
}

int multiply(){return 1;}

template<typename... Args> int multiply(int a,
    Args... arg){
  return (a * 1LL * multiply(arg...)) % MOD;
}
```

2 Combinatorial optimization

2.1 Lowest Common Ancestor

```
// 0-based vertex indexing. memset to -1
int log(int t){
  int res = 1;
  for(; 1 << res <= t; res++);
  return res;
}
int lca(int u , int v){
  if(h[u] < h[v])swap(u , v);
  int L = log(h[u]);
  for(int i = L - 1; i >= 0; i--){
    if(par[u][i] + 1 && h[u] - (1 << i) >= h[v])
      u = par[u][i];
  }
  if(v == u) return u;
  for(int i = L - 1; i >= 0; i--){
    if(par[u][i] + 1 && par[u][i] != par[v][i]){
      u = par[u][i]; v = par[v][i];
    }
  }
  return par[u][0];
}
```

2.2 Heavy-Light Decomposition

```
int n;
V<V<P<int>> > g;
V<int> values;
V<int> depth;
V<int> size;
V<int> hchild;
V<V<int> > chains;
V<int> head;
V<int> cmap,idmap;
V<int> parent;
V<P<int> > Edges;
V<Segtree> trees;
void dfs(int u,int p){
  parent[u]=p;
  depth[u]=depth[p]+1;
  size[u]=1;
  int hs=0,hv=-1;
  for(auto v:g[u])  if(v.fi!=p){
    dfs(v.fi,u);
    values[v.fi]=v.se;
    size[u]+=size[v.fi];
    if(size[v.fi]>hs){
      hv=v.fi;
      hs=size[v.fi];;
    }
  }
  hchild[u]=hv;
}
void form(int u,int p){
  cmap[u]=(int)chains.size()-1;
```

```
    idmap[u]=(int)chains.back().size();
    if(chains.back().size()==0){
      head.pb(u);
    }
    chains.back().pb(values[u]);
    if(hchild[u]!=-1)
      form(hchild[u],u);
    for(auto v:g[u])  if(v.fi!=p && v.fi!=hchild[u]) {
      chains.pb({});
      form(v.fi,u);
    }
}
void build(){
    for(int i=0;i<chains.size();i++){
      trees.pb(Segtree(chains[i]));
    }
}
int query(int u,int v){
    int r=0;
    while(u!=v){
      if(cmap[v]==cmap[u]){
        if(depth[v]<depth[u])
          swap(v,u);
        r=max(r,trees[cmap[v]].query(idmap[u]+1,idmap[v]));
        v=u;
      }
      else{
        if(depth[head[cmap[v]]]<depth[head[cmap[u]]])
          swap(v,u);
        int h=head[cmap[v]];
        int hid=idmap[h];
        int vid=idmap[v];
        r=max(r,trees[cmap[v]].query(hid,vid));
        v=parent[h];
      }
    }
    return r;
}
void update(int idx,int val){
    auto p=Edges[idx-1];
    int v=p.se;
    if(depth[p.se]<depth[p.fi])
      v=p.fi;
    trees[cmap[v]].update(idmap[v],val);
}
```

2.3 Auxiliary Tree

```

//std::vector<int> a contains vertices to form the aux t
sort(ALL(a), [](const int & a, const int & b) -> bool{
    return st[a] < st[b];
});
set<int> s(a);
for(int i = 0, k = (int)a.size(); i + 1 < k; i++){
    int v = lca(a[i], a[i + 1]);
    if(s.find(v) == s.end())
      a.push_back(v);
    s.insert(v);
}

sort(ALL(a), [](const int & a, const int & b) -> bool{
    return st[a] < st[b];
});

stack<int> S;
S.push(a[0]);

auto anc = [](int & a, int & b) -> bool{
    return st[b] >= st[a] && en[b] <= en[a];
};

for(int i = 1; i < (int)a.size(); i++){
    while(!anc(S.top(), a[i])) S.pop();
    G[S.top()].pp(a[i]);
    G[a[i]].pp(S.top());
    S.push(a[i]);
}
//G is the Aux tree
```

2.4 Articulation Point and Bridges

```
#include <bits/stdc++.h>

using namespace std;
const int N = 50;
```

```
int dis[N], low[N], par[N], AP[N],vis[N],tits;
void update(int u , int i, int child) {
  //For Cut Vertices
  if(par[u] != -1 && low[i] >= dis[u]) AP[u] = true;
  if(par[u] == -1 && child > 1) AP[u] = true;

  //For Finding Cut Bridge
  if(low[i] > dis[u]){
    //articulation bridge found.
  }
}
void dfs(int u){
  vis[u] = true;
  low[u] = dis[u] = (++tits); int child = 0;
  for(int i : g[u]) {
    if(!vis[i]){
      child++;
      par[i] = u;
      dfs(i);
      low[u] = min(low[u] , low[i]);
      update(u, i, child);
    }
    else if(i != par[u]) {
      low[u] = min(low[u] , dis[i]);
    }
  }
}
```

2.5 Biconnected Components

```
#include <bits/stdc++.h>
using namespace std;
const int N = (int)2e5 + 10;

vector<vector<int>> tree, g;
bool isBridge[N << 2], vis[N];
int Time, arr[N], U[N], V[N], cmpno, comp[N];
vector<int> temp; //temp stores component values

int adj(int u, int e){
  return (u == U[e] ? V[e] : U[e]);
}

int find_bridge(int u , int edge){
  vis[u] = true;
  arr[u] = Time++;
  int x = arr[u];

  for(auto & i : g[u]){
    int v = adj(u, i);
    if(!vis[v]){
      x = min(x,find_bridge(v, i));
    }
    else if(i != edge){
      x = min(x,arr[v]);
    }
  }

  if(x == arr[u] && edge != -1){
    isBridge[edge] = true;
  }
  return x;
}

void dfs1(int u){
  int current = cmpno;
  queue<int> q;
  q.push(u);
  vis[u] = 1;
  temp.push_back(current);

  while(!q.empty()){
    int v = q.front();
    q.pop();
    comp[v] = current;

    for(auto & i : g[v]) {
      int w = adj(v, i);
      if(vis[w])continue;
      if(isBridge[i]){
        cmpno++;
        tree[current].push_back(cmpno);
        tree[cmpno].push_back(current);
        dfs1(w);
      }
      else{
        q.push(w);
        vis[w] = 1;
      }
    }
  }
}
```

```
int main(){
  int n, m;
  cin >> n >> m;
  g.resize(n + 2); tree.resize(n + 2);

  for(int i = 0; i < m; i ++){
    cin >> U[i] >> V[i];
    g[U[i]].push_back(i);
    g[V[i]].push_back(i);
  }

  cmpno = Time = 0;
  memset(vis, false, sizeof vis);

  for(int i = 0; i < n; i ++){
    if(!vis[i]){
      find_bridge(i , -1);
    }
  }

  memset(vis, false, sizeof vis);
  cmpno = 0;

  for(int i = 0; i < n; i ++){
    if(!vis[i]){
      temp.clear();
      cmpno++;
      dfs1(i);
    }
  }
}
```

2.6 2-SAT

```
class sat_2{
public:
  int n, m, tag;
  V<V<int>> g, grev;
  V<bool> val;
  V<int> st;
  V<int> comp;

  sat_2(){}
  sat_2(int n) : n(n), m(2 * n), tag(0), g(m + 1),
      grev(m + 1), val(n + 1) { }

  void add_edge(int u, int v) { //u or v
    auto make_edge = [&](int a, int b) {
      if(a < 0) a = n - a;
      if(b < 0) b = n - b;
      g[a].pp(b);
      grev[b].pp(a);
    };

    make_edge(-u, v);
    make_edge(-v, u);
  }

  void truth_table(int u, int v, V<int> t) {
    for(int i = 0; i < 2; i ++) for(int j = 0; j <
        2; j ++) {
      if(!t[i * 2 + j])
        add_edge((2 * (i ^ 1) - 1) * u, (2 * (j ^
            1) - 1) * v);
    }
  }

  void dfs(int u, V<V<int>> & G, bool first) {
    comp[u] = tag;
    for(int & i : G[u]) if(comp[i] == -1)
      dfs(i, G, first);
    if(first) st.push_back(u);
  }

  bool satisfiable() {
    tag = 0; comp.assign(m + 1, -1);
    for(int i = 1; i <= m; i ++) {
      if(comp[i] == -1)
        dfs(i, g, true);
    }reverse(ALL(st));

    tag = 0; comp.assign(m + 1, -1);
    for(int & i : st) {
      if(comp[i] != -1) continue;
      tag++;
      dfs(i, grev, false);
    }

    for(int i = 1; i <= n; i ++) {
      if(comp[i] == comp[i + n]) return false;
      val[i] = comp[i] > comp[i + n];
    }
```

```

    return true;
  }
};
```

2.7 Dinic's Max Flow

```
// from stanford notebook
struct edge {
  int u, v;
  ll c, f;
  edge() { }
  edge(int _u, int _v, ll _c, ll _f = 0): u(_u), v(_v), c(_c), f(_f) { }
};
int n;
vector<edge> edges;
vector<vector<int> > g;
vector<int> d, pt;

void addEdge(int u, int v, ll c, ll f = 0) {
  g[u].emplace_back(edges.size());
  edges.emplace_back(edge(u,v,c,f));
  g[v].emplace_back(edges.size());
  edges.emplace_back(edge(v,u,0,0));
}
bool bfs(int s, int t) {
  queue<int> q({s});
  d.assign(n+1, n+2);
  d[s] = 0;
  while(!q.empty()) {
    int u = q.front(); q.pop();
    if (u == t) break;
    for(int k : g[u]) {
      edge &e = edges[k];
      if(e.f < e.c && d[e.v] > d[e.u] + 1){
        d[e.v] = d[e.u] + 1;
        q.push(e.v);
      }
    }
  }
  return d[t] < n+2;
}

ll dfs(int u, int t, ll flow = -1) {
  if(u == t || !flow) return flow;
  for(int &i = pt[u]; i < (int)(g[u].size()); i++) {
    edge &e = edges[g[u][i]], &oe=edges[g[u][i]^1];
    if(d[e.v] == d[e.u] + 1) {
      ll amt = e.c - e.f;
      if (flow != -1 && amt > flow) amt = flow;
      if(ll pushed = dfs(e.v,t,amt)) {
        e.f += pushed;
        oe.f -= pushed;
        return pushed;
      }
    }
  }
  return 0;
}

ll flow(int s, int t) {
  ll ans = 0;
  while(bfs(s,t)) {
    pt.assign(n+1, 0);
    while(ll val = dfs(s,t)) ans += val;
  }
  return ans;
}
```

2.8 Min Cost Max Flow

```
int tt=0;
class CostFlowGraph{
public:
  struct Edge{
    int v,f,c;
    Edge(){}
    Edge(int v,int f,int c):v(v),f(f),c(c){}
  };
  V<V<int> > g;
  V<Edge> e;
  V<int> pot;
  int n, flow, cost;
  CostFlowGraph(int sz){
    n=sz;
    g.resize(n);
    pot.assign(n,0);
    flow=0;
    cost=0;
```

```cpp
  }
  void clear(){
    flow=0;cost=0;
    for(int i=0;i<(int)e.size();i++){
      e[i].f+=e[i^1].f;
      e[i^1].f=0;
    }
  }
  void addEdge(int u,int v,int cap,int c){
    g[u].pb((int)e.size());
    e.pb(Edge(v,cap,c));
    g[v].pb((int)e.size());
    e.pb(Edge(u,0,-c));
  }
  void assignPots(int s){
    priority_queue<pii,V<pii>,greater<pii>> q;
    V<int> npot(n,inf);
    q.push({s,0});
    while(!q.empty()){
      auto cur=q.top();q.pop();
      if(npot[cur.fi]<=cur.se)
        continue;
      npot[cur.fi]=cur.se;
      for(auto i:g[cur.fi]) if(e[i].f>0){
        int cst=pot[cur.fi]-pot[e[i].v]+e[i].c;
        q.push({e[i].v,cst+cur.se});
      }
    }
    for(int i=0;i<n;i++)  if(npot[i]!=inf){
      pot[i]+=npot[i];
    }
  }
  void dfs(int t,V<bool> &v,V<int> &stk){
    auto cur=stk.back();
    v[e[cur].v]=1;
    if(e[stk.back()].v==t)
      return ;
    for(auto i:g[e[cur].v]) if(!v[e[i].v] && e[i].f>0 && (pot[e[cur].v]-pot[e[i].v]+e[i].c)==0){
      stk.pb(i);
      dfs(t,v,stk);
      if(e[stk.back()].v==t)
        return ;
    }
    stk.pop_back();
  }
  int augment(int s,int t){
    V<bool> v(n,false);
    vector<int> stk;
    if(g[s].size()==0)
      return 0;
    stk.pb(g[s][0]^1);
    dfs(t,v,stk);
    if(stk.empty())
      return 0;
    int mx=inf;
    for(int i=1;i<(int)stk.size();i++)
      mx=min(mx,e[stk[i]].f);
    for(int i=1;i<(int)stk.size();i++){
      e[stk[i]].f-=mx;
      e[(stk[i])^1].f+=mx;
    }
    return mx;
  }
  void mcf(int s,int t){
    int cur=0;
    do{
      flow+=cur;
      cost+=(pot[t]-pot[s]);
      assignPots(s);
      cur=augment(s,t);
    }while(cur);
  }
};
```

2.9 Global Min Cut

```cpp
// Adj mat. Stoer-Wagner min cut algorithm.
// Running time:O(|V|^3)
typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
```

```
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
  if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
  for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
  for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
  used[last] = true;
  cut.push_back(last);
  if (best_weight == -1 || w[last] < best_weight) {
    best_cut = cut;
    best_weight = w[last];
  }
      } else {
  for (int j = 0; j < N; j++)
    w[j] += weights[last][j];
  added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}

int main() {
  int N;
  cin >> N;
  for(int i = 0; i < N; i++) {
    int n, m;
    cin >> n >> m;
    VVI weights(n, VI(n));
    for (int j = 0; j < m; j++) {
      int a, b, c;
      cin >> a >> b >> c;
      weights[a-1][b-1] = weights[b-1][a-1] = c;
    }
    pair<int, VI> res = GetMinCut(weights);
    cout << "Case #" << i+1 << ": " << res.first << endl;
  }
}
```

2.10 Bipartite Matching

```
// maximum cardinality bipartite matching using augmenting paths.
// assumes that first n elements of graph adjacency list belong to the left vertex set.
int n;
vector<vector<int>> graph;
vector<int> match, vis;

int augment(int l) {
  if(vis[l]) return 0;
  vis[l] = 1;
  for(auto r: graph[l]) {
    if(match[r]==-1 || augment(match[r])) {
      match[r]=l; return 1;
    }
  }
  return 0;
}

int matching() {
  int ans = 0;
  for(int l = 0; l < n; l++) {
    vis.assign(n, 0);
    ans += augment(l);
  }
  return ans;
}
```

2.11 Hopcraft-Karp

```
#define MAX 100001
#define NIL 0
#define INF (1<<28)

vector< int > G[MAX];
int n, m, match[MAX], dist[MAX];
// n: number of nodes on left side, nodes are numbered 1 to n
// m: number of nodes on right side, nodes are numbered n+1 to n+m
```

```
// G = NIL[0]  ł  G1[G[1---n]]  ł  G2[G[n+1---n+m]]

bool bfs() {
    int i, u, v, len;
    queue< int > Q;
    for(i=1; i<=n; i++) {
        if(match[i]==NIL) {
            dist[i] = 0;
            Q.push(i);
        }
        else dist[i] = INF;
    }
    dist[NIL] = INF;
    while(!Q.empty()) {
        u = Q.front(); Q.pop();
        if(u!=NIL) {
            len = G[u].size();
            for(i=0; i<len; i++) {
                v = G[u][i];
                if(dist[match[v]]==INF) {
                    dist[match[v]] = dist[u] + 1;
                    Q.push(match[v]);
                }
            }
        }
    }
    return (dist[NIL]!=INF);
}

bool dfs(int u) {
    int i, v, len;
    if(u!=NIL) {
        len = G[u].size();
        for(i=0; i<len; i++) {
            v = G[u][i];
            if(dist[match[v]]==dist[u]+1) {
                if(dfs(match[v])) {
                    match[v] = u;
                    match[u] = v;
                    return true;
                }
            }
        }
        dist[u] = INF;
        return false;
    }
    return true;
}

int hopcroft_karp() {
    int matching = 0, i;
    // match[] is assumed NIL for all vertex in G
    while(bfs())
        for(i=1; i<=n; i++)
            if(match[i]==NIL && dfs(i))
                matching++;
    return matching;
}
```

2.12 Hungarian

```
//  Min cost BPM via shortest augmenting paths
//  O(n^3).Solves 1000x1000 in ~1s
//  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 with
//  The values in cost[i][j] may be +/-.  To perform
//  maximization, negate cost[][].
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);
  VD 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];
```

```cpp
    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;
```

```
}
```

3 Data Structures

3.1 Implicit Treap

```cpp
//1-based with lazy-updates, range sum query
struct node {
    int val, sum, lazy, prior, size;
    node *l, *r;
};
const int N = 2e5;
node pool[N]; int poolptr=0;
typedef node* pnode;
int sz(pnode t) { return t?t->size:0; }
void upd_sz(pnode t) { if(t) t->size = sz(t->l) + 1 + sz(t->r); }
void lazy(pnode t) {
    if(!t || !t->lazy) return;
    t->val+=t->lazy;
    t->sum+=t->lazy*sz(t);
    if(t->l)t->l->lazy+=t->lazy;
    if(t->r)t->r->lazy+=t->lazy;
    t->lazy = 0;
}
void reset(pnode t) {
    if(t) t->sum=t->val;
}
void combine(pnode& t, pnode l, pnode r) {
    if(!l || !r) return void(t=l?l:r);
    t->sum = l->sum + r->sum;
}
void operation(pnode t) {
    if(!t) return;
    reset(t);
    lazy(t->l); lazy(t->r);
    combine(t,t->l,t); combine(t,t,t->r);
}
void split(pnode t, pnode& l, pnode& r, int pos, int add = 0) {
    if(!t) return void(l=r=NULL);
    lazy(t); int curr_pos = add + sz(t->l);
    if(curr_pos<pos) split(t->r,t->r,r,pos,curr_pos+1),l=t;
    else split(t->l,l,t->r,pos,add),r=t;
    upd_sz(t); operation(t);
}
void merge(pnode& t, pnode l, pnode r) {
    lazy(l); lazy(r);
    if(!l || !r) t = l?l:r;
    else if(l->prior > r->prior) merge(l->r,l->r,r),t=l;
    else merge(r->l, l, r-> l), t=r;
    upd_sz(t); operation(t);
}
pnode init(int val) {
    pnode ret = &(pool[poolptr++]);
    ret->prior = rand(); ret->size = 1;
    ret->val = val; ret->sum = val; ret->lazy = 0;
    return ret;
}
int query(pnode t, int l, int r) {
    pnode L,mid,R;
    split(t, L, mid, l-1); split(mid, t, R, r-l);
    int ans = t->sum;
    merge(mid, L, t); merge(t, mid, R);
    return ans;
}
void upd(pnode t, int l, int r, int val) {
    pnode L, mid, R;
    split(t, L, mid, l-1); split(mid, t, R, r-l);
    t->lazy += val;
    merge(mid, L, t); merge(t, mid, R);
}
void insert(pnode& t, ll val, int pos) {
    pnode l;
    split(t,l,t,pos-1); merge(l,l,init(val)); merge(t,l,t);
}
```

3.2 Segment Tree

```cpp
// This code solves problem Help Ashu on hackerearth
// Iterative segment tree supporting non commutative combiner function
// The combiner function and identity of the combiner function are taken as contructor arguments
// Assign the initial input into t[size] to t[2*size-1] then call build
```

```
// Memory 2*size*sizeof(T)
// Time complexity O(log(size))
#include <bits/stdc++.h>
using namespace std;
/* Equinox */
template<typename T>
class SegTree{
public:
  vector<T> t;
  T identity;
  T (*combine)(T,T);
  int size;
  SegTree(T (*op)(T,T),T e,int n){
    combine=op;
    identity=e;
    t.assign(2*n,e);
    size=n;
  }
  void build(){for(int i=size-1;i>0;i--)t[i]=combine(t[i<<1],t[i<<1|1]);}
  T query(int l,int r){
    T lt=identity;
    T rt=identity;
    for(l+=size,r+=size;l<=r;r>>=1,l>>=1){
      if(l&1)   lt=combine(lt,t[l++]);
      if(!(r&1))  rt=combine(t[r--],rt);
    }
    return combine(lt,rt);
  }
  void update(int p,T v){for(t[p+=size]=v;p>>=1;)t[p]=combine(t[p<<1],t[p<<1|1]);}
};
int32_t main(){
  int n;
  cin>>n;
  SegTree<int> tree([](int a,int b){return a+b;},0,n);
  for(int i=0;i<n;i++){
    int a;
    cin>>a;
    tree.t[i+n]=a&1;
  }
  tree.build();
  int q;
  cin>>q;
  while(q--){
    int c,x,y;
    cin>>c>>x>>y;
    switch(c){
      case 0:
      tree.update(x-1,y&1);
      break;
      case 1:
      cout<<(y-x+1)-tree.query(x-1,y-1)<<"\n";
      break;
      case 2:
      cout<<tree.query(x-1,y-1)<<"\n";
    }
  }
  return 0;
}
```

3.3 Lazy Propagation

```
// This code solves problem LITE on spoj
// Iterative segment tree with lazy propagation supporting non commutative combiner functions
// The combiner function and identity of the combiner function are taken as contructor arguments
// Also the function for application of lazy nodes onto tree nodes is taken as parameter along with Zero of lazy node
// Assign the initial input into t[size] to t[2*size-1] then call build
// Memory 2*size*sizeof(T)+2*size*sizeof(L)
// Time complexity O(log(size))
#include <bits/stdc++.h>
using namespace std;
/* Equinox */
template<typename T,typename L>
class SegTree{
public:
  vector<T> t;
  vector<T> lz;
  T identity;
  L zero;
  T (*combine)(T,T);
  void (*apply)(T&,L&,L&,int k);
  int size;
  int height;
```

```
SegTree(T (*op)(T,T),T e,void (*pro)(T&,L&,L&,int k),L z,int n){
  combine=op;
  apply=pro;
  identity=e;
  zero=z;
  t.assign(2*n,e);
  lz.assign(n,z);
  size=n;
  height = sizeof(int)*8-__builtin_clz(n);
}
void build(){for(int i=size-1;i>0;i--)t[i]=combine(t[i<<1],t[i<<1|1]);}
void push(int p){
  for(int s=height;s>0;s--){
    int i=p>>s;
    apply(t[i<<1],lz[i<<1],lz[i],1<<(s-1));
    apply(t[i<<1|1],lz[i<<1|1],lz[i],1<<(s-1));
    lz[i]=zero;
  }
}
void reassign(int p){
  for(p>>=1;p>0;p>>=1)
    if(lz[p]==zero)
      t[p]=combine(t[p<<1],t[p<<1|1]);
}
T query(int l,int r){
  push(l+=size);
  push(r+=size);
  T lt=identity;
  T rt=identity;
  for(;l<=r;r>>=1,l>>=1){
    if(l&1)   lt=combine(lt,t[l++]);
    if(!(r&1))  rt=combine(t[r--],rt);
  }
  return combine(lt,rt);
}
void update(int p,T v){push(p);for(t[p+=size]=v;p>>=1;)t[p]=combine(t[p<<1],t[p<<1|1]);}
void update(int l,int r,L v){
  push(l+=size);
  push(r+=size);
  int k=1;
  int l0=l,r0=r;
  for(;l<=r;r>>=1,l>>=1,k<<=1){
    if(l&1)   apply(t[l],lz[l],v,k),l++;
    if(!(r&1))  apply(t[r],lz[r],v,k),r--;
  }
  reassign(l0);
  reassign(r0);
 }
};
int32_t main(){
  int n,m;
  cin>>n>>m;
  SegTree<int,int> s([](int a, int b){return a + b;},0,[](int &v,int &l,int &u,int k){if(u)v=k-v;l^=u;},0,n);
  while(m--){
    int c;
    cin>>c;
    if(!c){
      int l,r;
      cin>>l>>r;
      s.update(l-1,r-1,1);
    }
    else{
      int l,r;
      cin>>l>>r;
      cout<<s.query(l-1,r-1)<<"\n";
    }
  }
  return 0;
}
```

3.4 Disjoin Set Union

```
class dsu{
  public:
  vector<int> p;
  dsu(int n){
    p.resize(n);
    for(int i=0;i<n;i++)
      p[i]=i;
  }
  int parent(int x){
    return x==p[x]?x:x=parent(p[x]);
  }
  void unite(int x,int y){
    x=parent(x);
```

```
    y=parent(y);
    if(x==y)
      return;
    p[x]=y;
  }
  bool check(int x,int y){
    x=parent(x);
    y=parent(y);
    return x==y;
  }
};
```

4 Math

4.1 Extended Euclid

```
#include <bits/stdc++.h>

using namespace std;
using LL = long long;

template<typename T> T gcd(T a , T b){return (a ?
   gcd(b % a , a): b);} //supposing a is small and
    b is large.
template<typename T> pair<T,T> extend_euclid(T a,
   T b){ //supposing a is small and b is large.
  pair<T,T> a_one = {1, 0} , b_one = {0 , 1};
  // b_one is just the second last step's
     coefficient, a_one is the last step's
     coefficient
  if(!b)return a_one;
  while(a){
    /* We first start from writing
    b = 0(a) + 1(b), for which it's b_one
    a = 1(a) + 0(b), for which it's a_one
    b = b % a + (b / a)*a, then
    */
    T q = b / a; T r = b % a;
    T dx = b_one.first - q*a_one.first;
    T dy = b_one.second - q*a_one.second;
    b = a; a = r;
    b_one = a_one;
    a_one = {dx , dy};
  }
  return b_one;
}

int main(){
  LL a, m; cin >> a >> m;
  auto ans = extend_euclid(a, m);
  LL x = (ans.first + m)%m; //Inverse Modulo (m) $
      ax=1 mod(m) and gcd(a,m) == 1
  cout << (ans.first + m) % m << endl;
  return 0;
}
```

4.2 Fast Exponentiation

```
// Takes a base 'b', exponent 'e' and modulo 'm'
// m should be less than 3e9
// Complexity O(log(e))
#define ll long long
ll modex(ll b,ll e,ll m){
  ll r=1;
  b%=m;
  while(e){
    if(e&1) r=(r*b)%m;
    e>>=1;
    b=(b*b)%m;
  }
  return r;
}
```

4.3 Fast Fourier Transform

```
const long double PI=acos(-1.0);
typedef long long ll;
typedef long double ld;
typedef vector<ll> VL;
int bits(int x){
  int r=0;
  while(x){
    r++;
    x>>=1;
  }
  return r;
}
int reverseBits(int x,int b){
  int r=0;
  for(int i=0;i<b;i++){
    r<<=1;
```

```
    r|=(x&1);
    x>>=1;
  }
  return r;
}
class Complex{
  public:
  ld r,i;
  Complex(){r=0.0;i=0.0;}
  Complex(ld a,ld b){r=a;i=b;}
};
Complex operator*(Complex a,Complex b){
  return Complex(a.r*b.r-a.i*b.i,a.r*b.i+a.i*b.r);
}
Complex operator-(Complex a,Complex b){
  return Complex(a.r-b.r,a.i-b.i);
}
Complex operator+(Complex a,Complex b){
  return Complex(a.r+b.r,a.i+b.i);
}
Complex operator/(Complex a,ld b){
  return Complex(a.r/b,a.i/b);
}
Complex EXP(ld theta){
  return Complex(cos(theta),sin(theta));
}

typedef vector<Complex> VC;

void FFT(VC& A,int inv){
  int l=A.size();
  int b=bits(l)-1;
  VC a(A);
  for(int i=0;i<l;i++){
    A[reverseBits(i,b)]=a[i];
  }
  for(int i=1;i<=b;i++){
    int m=(1<<i);
    int n=m>>1;
    Complex wn=EXP((ld)inv*(ld)2.0*PI/(ld)m);
    for(int j=0;j<l;j+=m){
      Complex w(1.0,0.0);
      for(int k=j;k<j+n;k++){
        Complex t1=A[k]+w*A[k+n];
        Complex t2=A[k]-w*A[k+n];
        A[k]=t1;
        A[k+n]=t2;
        w=w*wn;
      }
    }
  }
  if(inv==-1){
    for(auto &i:A){
      i=i/(ld)l;
    }
  }
}

VL Convolution(VL & a,VL & b){
  int tot_size = (int)a.size() + (int)b.size();
  int bit = bits(tot_size);
  int l = 1 << bit;
  VC A, B, C;
  A.reserve(l); B.reserve(l); C.reserve(l);
  for(int i = 0; i < l; i ++) {
    if(i < (int)a.size()) A.pb({(ld)a[i], 0.0});
    else A.pb({0.0, 0.0});
    if(i < (int)b.size()) B.pb({(ld)b[i], 0.0});
    else B.pb({0.0, 0.0});
  }
  FFT(A, 1);
  FFT(B, 1);
  for(int i = 0; i < l; i ++) {
    C.pb(A[i] * B[i]);
  }
  FFT(C, -1);
  VL c;
  for(auto & i : C) {
    c.pb(round(i.r));
  }
  return c;
}
```

4.4 Large Factorial

```
ll fmod(ll x,ll md,ll p){
  V<ll> pre(md);
  pre[0]=1;
  for(ll i=1;i<md;i++){
    if(i%p!=0)
      pre[i]=(pre[i-1]*i)%md;
    else
```

```
      pre[i]=pre[i-1];
  }
  ll r=1;
  while(x){
    ll cy=x/md;
    r=(r*modex(pre[md-1],cy,md))%md;
    r=(r*pre[x%md])%md;
    x/=p;
  }
  return r;
}
```

4.5 Large Modulo Exponentiation

```
// Uses LargeModuloMultiplication for fast
   exponentiation
// m can be as big as 10^18
// Time complexity O(log(e))
#define ll long long
#define ld long double
ll mulmod(ll a, ll b, ll m){
  ll q = (ll)(((ld)a*(ld)b) / (ld)m);
  ll r = a*b - q*m;
  if (r > m)r %= m;
  if (r < 0)r += m;
  return r;
}
ll modex(ll b,ll e,ll m){
  ll r=1;
  b%=m;
  while(e){
    if(e&1) r=mulmod(r,b,m);
    e>>=1;
    b=mulmod(b,b,m)%m;
  }
  return r;
}
```

4.6 Large Modulo Multiplication

```
// Finds (a*b)%m when either can be as big as
   10^18
#define ll long long
#define ld long double
ll mulmod(ll a, ll b, ll m){
  a%=m;b%=m;
  ll q = (ll)(((ld)a*(ld)b) / (ld)m);
  ll r = a*b - q*m;
  if (r > m)r %= m;
  if (r < 0)r += m;
  return r;
}
```

4.7 Segmented Sieve

```
// Segmented Seive
// N=sqrt(b)
// Time complexity: O(N log(B-A))
#define A 1000000000000LL
#define B 1000000100000LL
bitset<B-A> p;
void seive(){
  p.set();
  for(ll i=2;i*i<=B;i++){
    for(ll j=((A+i-1)/i)*i;j<=B;j+=i){
      p.reset(j-A);
    }
  }
}
```