University of Haifa ICPC Team Notebook 2017

Contents

1 Miscellaneous

1.1 Template

```
#include <bits/stdc++.h>
#define loop(i, a, n) for (int i = a; i < int(n); ++i)
#define loop_rev(i, b, a) for (ll i = b; i >= ll(a); --i)
using namespace std;
typedef long long ll;
typedef long double ld;
typedef pair<int,int> ii;
typedef vector<int> vi;
typedef vector<vi> vvi;

void solve(){

}

int main(){
        int t; cin >> t;
        while(t--)
                solve();
        return 0;
}
```

1.2 Input/Output c++

```
int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

1.3 Random STL stuff

```
// Example for using stringstreams and next_permutation

int main(void){
  vector<int> v;

  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);

  // Expected output: 1 2 3 4
  //                  1 2 4 3
  //                  ...
  //                  4 3 2 1
  do {
    ostringstream oss;
    oss << v[0] << " " << v[1] << " " << v[2] << " " << v[3];

    // for input from a string s,
    //   istringstream iss(s);
    //   iss >> variable;

    cout << oss.str() << endl;
  } while (next_permutation (v.begin(), v.end()));

  v.clear();

  v.push_back(1);
  v.push_back(2);
  v.push_back(1);
  v.push_back(3);

  // To use unique, first sort numbers.  Then call
  // unique to place all the unique elements at the beginning
  // of the vector, and then use erase to remove the duplicate
  // elements.

  sort(v.begin(), v.end());
  v.erase(unique(v.begin(), v.end()), v.end());

  // Expected output: 1 2 3
  for (size_t i = 0; i < v.size(); i++)
    cout << v[i] << " ";
  cout << endl;
}
```

2 Geometry

2.1 Geometry Vectors

```
typedef double T;
typedef pair<T, T> ii;
typedef pair<ii, T> i3;
typedef pair<ii, ii> i4;

inline double cross(ii a, ii b){
    return a.first * b.y - a.y * b.first;
}

inline ii operator+(ii a, ii b){
    return ii(a.first + b.first, a.y + b.y);
```

```
}

inline ii operator-(ii a, ii b){
    return ii(a.first - b.first, a.y - b.y);
}

inline double operator*(ii a, ii b){
    return a.first * b.first + a.y * b.y;
}

inline ii operator/(ii a, double b){
    return ii(a.first / b, a.y / b);
}

inline T norm(ii x){
    return x * x;
}

inline int sig(double x){
    return x > 0 ? 1 : x == 0 ? 0 : -1;
}

inline double distance(ii a, ii b){
    return sqrt(norm(b - a));
}
```

2.2 Convex Hull

```
inline int lowest_point(vector<ii>& P){
        return min_element(P.begin(), P.end(), [](ii a, ii b){return ii(a.second, a.first) < ii(b.
              second, b.first);}) - P.begin();
}

vector<ii> convex_hull(vector<ii> S){
    int first = lowest_point(S);
    ii origin = S[first];
    S.erase(S.begin() + first);
    sort(S.begin(), S.end(), [&origin](ii p1, ii p2){return ii(-cross(p1 - origin, p2 - origin), norm(
          p1 - origin)) < ii(0, norm(p2 - origin));});

    vector<ii> hull = {origin};
    for (auto& p : S){
        while(hull.size() >= 2 && cross(hull[hull.size() - 1] - hull[hull.size() - 2], p - hull[hull.
              size() - 1]) <= 0) hull.pop_back();
        hull.push_back(p);
    }
    return hull;
}
```

2.3 Minimal Bounding Circle

```
bool inside_circle(i3 circle, ii p){
    return distance(circle.first, p) <= circle.second;
}

i3 circle_from_2_points(ii a, ii b){
    return {(a+b)/2, distance(a, b)/2};
}

i3 circle_from_3_points(ii a, ii b, ii c){
    double d = 2.0*(a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y));
    double xc = ((a.x * a.x + a.y * a.y) * (b.y - c.y) + (b.first * b.first + b.y * b.y) * (c.y - a.y)
          + (c.first * c.first + c.y * c.y) * (a.y - b.y) ) / d;
    double yc = ((a.first * a.first + a.y * a.y) * (c.first - b.first) + (b.first * b.first + b.y * b.
          y) * (a.first - c.first) + (c.first * c.first + c.y * c.y) * (b.first - a.first) ) / d;
    cerr << xc << " " << yc << endl;
    ii center = {xc, yc};
    return {center, distance(center, a)};
}

i3 minimal_bounding_circle_3(vector<ii>& P, ii a, ii b){
    i3 circle = circle_from_2_points(a, b);
    for(ii p : P){
        if(!inside_circle(circle, p)){
            circle = circle_from_3_points(a, b, p);
        }
    }
    return circle;
}

i3 minimal_bounding_circle_2(vector<ii>& P, ii a){
    i3 circle = circle_from_2_points(a, P[0]);
    vector<ii> P2 = {P[0]};
    for(int i = 1; i < (int)P.size(); ++i){
        ii p = P[i];
        if(!inside_circle(circle, p)){
            circle = minimal_bounding_circle_3(P2, a, p);
        }
        P2.push_back(P[i]);
    }
    return circle;
}

i3 minimal_bounding_circle(vector<ii>& P){
    if(P.size() < 2) return {{0,0}, -1}; // null!
    random_shuffle(P.begin(), P.end());

    i3 circle = circle_from_2_points(P[0], P[1]);
    vector<ii> P2 = {P[0], P[1]};
    for(int i = 2; i < (int)P.size(); ++i){
        ii p = P[i];
        if(!inside_circle(circle, p)){
            circle = minimal_bounding_circle_2(P2, p);
        }
        P2.push_back(P[i]);
    }
    return circle;
}
```

3 Numerical algorithms

3.1 Gauss Elimination

```
const int mod = 2;

inline int mul(int a, int b){
    return (ll(a) * ll(b)) % mod;
}

int power(int a, int b){
    if (b == 0) return 1;
    int c = power(a, b / 2);
    return mul(mul(c, c), (b % 2 == 0 ? 1 : a));
}

inline int inv(int x){
    return power(x, mod - 2);
}

vector<int> operator-(vector<int> arr1, const vector<int>& arr2){
    loop(i, 0, arr1.size()) arr1[i] = (ll(arr1[i]) - ll(arr2[i]) + mod) % mod;
    return arr1;
}

vector<int> operator*(int k, vector<int> arr){
    for (auto& x : arr) x = (ll(x) * ll(k)) % mod;
    return arr;
}

#define N 2501
#define bool_array bitset<N>
const bool_array zero;

template<typename T>
void gauss(vector<T>& arr);

bool_array operator-(const bool_array& arr1, const bool_array& arr2){
    return arr1 ^ arr2;
}

bool_array operator*(int k, const bool_array& arr){
    return k == 1 ? arr : zero;
}

bool_array vector_int_to_bool_array(const vector<int>& arr){
    bool_array ret;
    loop(i, 0, arr.size()) ret[i] = arr[i];
    return ret;
}

vector<int> bool_array_to_vector_int(const bool_array& arr, int n){
    vector<int> ret(n);
    loop(i, 0, ret.size()) ret[i] = arr[i];
    return ret;
}

void gauss_com(vector<vector<int>>& arr){
```

```
    int m = arr.front().size();
    vector<bool_array> com(arr.size());
    loop(i, 0, com.size()) com[i] = vector_int_to_bool_array(arr[i]);
    gauss(com);
    arr.resize(com.size());
    loop(i, 0, com.size()) arr[i] = bool_array_to_vector_int(com[i], m);
}

template<typename T>
void gauss(vector<T>& arr){
    int row = 0;
    if (arr.size() == 0) return;
    loop(j, 0, arr[row].size()){
        int pos = -1;
        loop(k, row, arr.size()) if (arr[k][j] != 0){
            pos = k;
            break;
        }
        if (pos == -1) continue;
        swap(arr[pos], arr[row]);
        loop(k, row + 1, arr.size()) if (arr[k][j] != 0) arr[k] = arr[row][j] * arr[k] - arr[k][j] *
            arr[row];
        if (++row == (int)arr.size()) break;
    }

    while (arr.size() != 0){
        int b = true;
        loop(j, 0, arr.back().size()) if (arr.back()[j] != 0){
            b = false;
            break;
        }
        if (!b) break;
        arr.pop_back();
    }
    int col = -1;
    loop(i, 0, arr.size()){
        while (arr[i][++col] == 0);
        loop(k, 0, i) if (arr[k][col] != 0) arr[k] = arr[i][col] * arr[k] - arr[k][col] * arr[i];
    }
}
```

3.2 Fast Fourier Transform

```
#define pi 3.14159265359

typedef complex<double> com;

void dft(vector<com>& p, com mult){
    if (p.size() == 1) return;

    vector<com> p1(p.size() / 2), p2(p.size() / 2);
    loop(i, 0, p.size())
        if (i % 2 == 0) p1[i / 2] = p[i];
        else            p2[i / 2] = p[i];

    com curr = 1, new_mult = mult * mult;
    dft(p1, new_mult), dft(p2, new_mult);
    loop(i, 0, p.size() / 2){
        com a = p1[i], b = curr * p2[i];
        p[i] = a + b, p[i + p.size() / 2] = a - b;
        curr *= mult;
    }
}

void dft(vector<com>& p, int k){
    dft(p, polar(1., k * 2 * pi / p.size()));
}

vector<double> mul(const vector<double>& _p1, const vector<double>& _p2){
    vector<com> p1(_p1.size()), p2(_p2.size());
    loop(i, 0, p1.size()) p1[i] = _p1[i];
    loop(i, 0, p2.size()) p2[i] = _p2[i];

    int k = 1;
    while (k < 2 * (int)p1.size() - 1 || k < 2 * (int)p2.size() - 1) k *= 2;
    while ((int)p1.size() < k) p1.push_back(0);
    while ((int)p2.size() < k) p2.push_back(0);

    dft(p1, 1), dft(p2, 1);

    vector<com> p(p1.size());
    loop(i, 0, p1.size()) p[i] = p1[i] * p2[i];

    dft(p, -1);

    while (p.size() > 1 && norm(p.back()) < 0.001) p.pop_back();
    vector<double> res(p.size());
    loop(i, 0, res.size()) res[i] = real(p[i]) / p1.size();
    return res;
}
```

3.3 Hadamard Transform

```
const unsigned int mod = 1'000'000''007;

void trans(vi::iterator begin, vi::iterator end, int counter){
    if (counter == 0) return;
    int k = (end - begin) / 2;

    trans(begin, begin + k, counter - 1), trans(begin + k, end, counter - 1);
    loop(i, 0, k){
        unsigned int x = *(begin + i), y = *(begin + k + i);
        *(begin + i) = (x + y) % mod, *(begin + k + i) = (x + mod - y) % mod;
    }
}

vi mul(vi p1, vi p2, bool same = false){
    int k, counter;
    for (k = 1, counter = 0; k < (int)max(p1.size(), p2.size()); k *= 2, ++counter);
    p1.resize(k), p2.resize(k);

    trans(p1.begin(), p1.end(), counter);
    if (!same) trans(p2.begin(), p2.end(), counter);
    else p2 = p1;
    loop(i, 0, p1.size()) p1[i] = (ll(p1[i]) * ll(p2[i])) % mod;
    trans(p1.begin(), p1.end(), counter);

    int curr = 1, mul = mod / 2 + 1;
    loop(i, 0, counter) curr = (ll(curr) * ll(mul)) % mod;
    for (auto& x : p1) x = (ll(x) * ll(curr)) % mod;
    return p1;
}

vi power(vi p, int k){
    vi m = p;
    p.assign(p.size(), 0); p[0] = 1;
    while (k){
        if (k % 2 == 1) mul(p, m);
        k /= 2;
        mul(m, m, true);
    }
    return p;
}

int32_t main(){
    ios::sync_with_stdio(false);
    int k, l; cin >> k >> l;
    vi p(l + 1, 1);
    p[0] = p[1] = 0;
    loop(i, 2, p.size()) for (int j = 2 * i; j < (int)p.size(); j += i) p[j] = 0;

    p = power(p, k);
    cout << p[0] << endl;
    return 0;
}
```

3.4 Fast Subset Convolution

```
const unsigned int mod = 1'000'000''007;

void trans(vi& p){
    int n = p.size();
    for(int c = 1; c < n; c+=c)
    loop(i, 0, n)
    if(i & c)
        p[i] += p[i-c];
}

void invtrans(vi& p){
    int n = p.size();
    for(int c = n/2; c; c /=2)
    for(int i = n-1; i>= 0; i--)
    if(i & c)
        p[i] -= p[i-c];
}

vi mul(vi p1, vi p2){
    int n = p1.size();

    //loop(j, 0, n) cerr << p1[j]; cerr << endl;
    //loop(j, 0, n) cerr << p2[j]; cerr << endl;
```

```
    //cerr << endl;
    trans(p1);
    //loop(j, 0, n) cerr << p1[j]; cerr << endl;
    trans(p2);
    //loop(j, 0, n) cerr << p2[j]; cerr << endl;

    loop(i, 0, n) p1[i] *= p2[i];

    invtrans(p1);

    //loop(j, 0, n) cerr << p1[j]; cerr << endl;
    //cerr << endl;
    return p1;
}


int32_t main(){
    ios::sync_with_stdio(false);

    int n, m, k; cin >> n >> m >> k;
    vvi A(m, vi(1 << n));
    loop(i, 0, m)
    loop(j, 0, (1 << n)){
        char c; cin >> c;
        A[i][j] = (c == '1');
    }
    loop(i, 0 , k){
        int a,b ; cin >> a >> b;
        vi ans = mul(A[a], A[b]);
        loop(j, 0 , (1 << n))
            cout << (ans[j] ? 1 : 0);
        cout << endl;
    }


    return 0;
}

/*

3 5 3
11111010
11000000
11001000
11101000
11101000
0 1
1 2
3 4


*/
```

4 Graph algorithms

4.1 Strongly Connected Components

```
//Not Tested
#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x)
{
  int i;
  v[x]=true;
  for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
  stk[++stk[0]]=x;
}
void fill_backward(int x)
{
  int i;
  v[x]=false;
  group_num[x]=group_cnt;
  for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
}
void add_edge(int v1, int v2) //add edge v1->v2
{
  e [++E].e=v2; e [E].nxt=sp [v1]; sp [v1]=E;
  er[  E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC()
{
  int i;
  stk[0]=0;
  memset(v, false, sizeof(v));
  for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
  group_cnt=0;
  for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++; fill_backward(stk[i]);}
}
```

4.2 Dijkstra

```
vector<int> dijkstra(const graph_w& G, int s){
    int n = G.size();
    vector<int> dis(n, inf);
    set<ii> S;
    dis[s] = 0;
    S.insert({0, s});
    while(!S.empty()){
        int u = S.begin()->second;
        S.erase(S.begin());

        for(auto& e : G[u]){
            int v = e.first, w = e.second;
            if(dis[v] > dis[u] + w){
                S.erase({dis[v], v});
                dis[v] = dis[u] + w;
                S.insert({dis[v], v});
            }
        }
    }
    return dis;
}
```

4.3 Dinic Max Flow

```
//
// Dinic algorithm for maximum flow / minimum cut
// time: O(VVE), usually faster, no more than O(maxflow * E)
// space: O(V+E)
//
struct edge {
  int u, v;
  ll cap, flow;
  edge() {}
  edge(int u, int v, ll cap): u(u), v(v), cap(cap), flow(0) {}
};

struct Dinic {
  int N;
  vector<edge> E;
  vector<vector<int> > g;
  vector<int> d, pt;

  Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

  void Addedge(int u, int v, ll cap) {
    if (u != v) {
      E.emplace_back(edge(u, v, cap));
      g[u].emplace_back(E.size() - 1);
      E.emplace_back(edge(v, u, 0));
      g[v].emplace_back(E.size() - 1);
    }
  }

  bool BFS(int S, int T) {
    queue<int> q({S});
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while(!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k: g[u]) {
        edge &e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.emplace(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }
```

```
  ll DFS(int u, int T, ll flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int &i = pt[u]; i < g[u].size(); ++i) {
      edge &e = E[g[u][i]];
      edge &oe = E[g[u][i]^1];
      if (d[e.v] == d[e.u] + 1) {
        ll amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (ll pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }

  ll MaxFlow(int S, int T) {
    ll total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (ll flow = DFS(S, T))
        total += flow;
    }
    return total;
  }
};
```

4.4 Maximum Bipartite Matching

```
//Not Tested

// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

bool FindMatch(int i, const vvi &w, vi &mr, vi &mc, vi &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const vvi &w, vi &mr, vi &mc) {
  mr = vi(w.size(), -1);
  mc = vi(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    vi seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

4.5 Dominators

```
namespace dominator{
    int T, n;
    vvi g, rg, bucket;
    vi sdom, par, dom, dsu, label, arr, rev;

    int find(int u, int x = 0){
        if (u == dsu[u]) return x ? -1 : u;
        int v = find(dsu[u], x + 1);
        if (v < 0) return u;
        if (sdom[label[dsu[u]]] < sdom[label[u]]) label[u] = label[dsu[u]];
        dsu[u] = v;
        return x ? v : label[u];
    }

    void unite(int u, int v){
        dsu[v] = u;
    }

    void dfs(int u){
        T++; arr[u] = T; rev[T] = u;
        label[T] = T; sdom[T] = T; dsu[T] = T;
        loop(i, 0, g[u].size()){
            int w = g[u][i];
            if (!arr[w]) dfs(w), par[arr[w]] = arr[u];
            rg[arr[w]].PB(arr[u]);
        }
    }

    static vi get(const graph& G, int root){
        T = 0, n = G.size();
        int k = n + 1;
        g.assign(k, {}), rg.assign(k, {}), bucket.assign(k, {});
        sdom.assign(k, 0), par.assign(k, 0), dom.assign(k, 0), dsu.assign(k, 0), label.assign(k, 0),
              arr.assign(k, 0), rev.assign(k, 0);

        loop(i, 0, G.size()) for (auto& x : G[i]) g[i + 1].PB(x + 1);
        ++root;
        vector<int> res(n, -1);
        dfs(root); n = T;
        loop_rev(i, n, 1){
            loop(j, 0, rg[i].size()) sdom[i] = min(sdom[i], sdom[find(rg[i][j])]);
            if (i > 1) bucket[sdom[i]].PB(i);
            loop(j, 0, bucket[i].size()){
                int w = bucket[i][j],v = find(w);
                if (sdom[v] == sdom[w]) dom[w] = sdom[w];
                else dom[w] = v;
            }
            if (i > 1) unite(par[i], i);
        }
        loop(i, 2, n + 1){
            if (dom[i] != sdom[i]) dom[i] = dom[dom[i]];
            res[rev[i] - 1] = rev[dom[i]] - 1;
        }
        return res;
    }
}
```

5 Data structures

5.1 Disjoint Sets (Union-Find)

```
//Not Tested

vector<int> parent;

void init(int n){
        parent.resize(n);
        for(int i = 0;i<n;i++) parent[i] = i;
}

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

void uni(int u, int v){
        parent[find(u)] = find(v);
}
```

5.2 Fenwick Tree

```
//Not Tested
#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
  while(x <= N) {
    tree[x] += v;
    x += (x & -x);
  }
}
```

```
// get cumulative sum up to and including x
int get(int x) {
  int res = 0;
  while(x) {
    res += tree[x];
    x -= (x & -x);
  }
  return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
  int idx = 0, mask = N;
  while(mask && idx < N) {
    int t = idx + mask;
    if(x >= tree[t]) {
      idx = t;
      x -= tree[t];
    }
    mask >>= 1;
  }
  return idx;
}
```

5.3 Splay Tree

```
// Not Tested

const int N_MAX = 130010;
const int oo = 0x3f3f3f3f;
struct Node
{
  Node *ch[2], *pre;
  int val, size;
  bool isTurned;
} nodePool[N_MAX], *null, *root;

Node *allocNode(int val)
{
  static int freePos = 0;
  Node *x = &nodePool[freePos ++];
  x->val = val, x->isTurned = false;
  x->ch[0] = x->ch[1] = x->pre = null;
  x->size = 1;
  return x;
}

inline void update(Node *x)
{
  x->size = x->ch[0]->size + x->ch[1]->size + 1;
}

inline void makeTurned(Node *x)
{
  if(x == null)
    return;
  swap(x->ch[0], x->ch[1]);
  x->isTurned ^= 1;
}

inline void pushDown(Node *x)
{
  if(x->isTurned)
  {
    makeTurned(x->ch[0]);
    makeTurned(x->ch[1]);
    x->isTurned ^= 1;
  }
}

inline void rotate(Node *x, int c)
{
  Node *y = x->pre;
  x->pre = y->pre;
  if(y->pre != null)
    y->pre->ch[y == y->pre->ch[1]] = x;
  y->ch[!c] = x->ch[c];
  if(x->ch[c] != null)
    x->ch[c]->pre = y;
  x->ch[c] = y, y->pre = x;
  update(y);
  if(y == root)
    root = x;
}

void splay(Node *x, Node *p)
{
  while(x->pre != p)
  {
    if(x->pre->pre == p)
      rotate(x, x == x->pre->ch[0]);
    else
    {
      Node *y = x->pre, *z = y->pre;
      if(y == z->ch[0])
      {
        if(x == y->ch[0])
          rotate(y, 1), rotate(x, 1);
        else
          rotate(x, 0), rotate(x, 1);
      }
      else
      {
        if(x == y->ch[1])
          rotate(y, 0), rotate(x, 0);
        else
          rotate(x, 1), rotate(x, 0);
      }
    }
  }
  update(x);
}

void select(int k, Node *fa)
{
  Node *now = root;
  while(1)
  {
    pushDown(now);
    int tmp = now->ch[0]->size + 1;
    if(tmp == k)
      break;
    else if(tmp < k)
      now = now->ch[1], k -= tmp;
    else
      now = now->ch[0];
  }
  splay(now, fa);
}

Node *makeTree(Node *p, int l, int r)
{
  if(l > r)
    return null;
  int mid = (l + r) / 2;
  Node *x = allocNode(mid);
  x->pre = p;
  x->ch[0] = makeTree(x, l, mid - 1);
  x->ch[1] = makeTree(x, mid + 1, r);
  update(x);
  return x;
}

int main()
{
  int n, m;
  null = allocNode(0);
  null->size = 0;
  root = allocNode(0);
  root->ch[1] = allocNode(oo);
  root->ch[1]->pre = root;
  update(root);

  scanf("%d%d", &n, &m);
  root->ch[1]->ch[0] = makeTree(root->ch[1], 1, n);
  splay(root->ch[1]->ch[0], null);

  while(m --)
  {
    int a, b;
    scanf("%d%d", &a, &b);
    a ++, b ++;
    select(a - 1, null);
    select(b + 1, root);
    makeTurned(root->ch[1]->ch[0]);
  }

  for(int i = 1; i <= n; i ++)
  {
    select(i + 1, null);
    printf("%d ", root->val);
  }
}
```

5.4 Convex Hull Trick

```
typedef pair<ll, ll> llll;

enum line_type{
    normal,
    minus_infinity,
    infinity,
    value
};

ll gcd(ll a, ll b){
    if (a < b) swap(a, b);
    if (b == 0) return a;
    return gcd(b, a % b);
}

struct fraction{
    ll a, b;
    fraction(ll _a, ll _b = 1) : a(_a), b(_b){
        if (a == 0 && b == 0) return;
        if (b < 0) a = -a, b = -b;
        ll c = gcd(abs(a), b);
        a /= c, b /= c;
    }

    operator llll() const{
        return llll(a, b);
    }
};

inline bool operator<(const fraction& f1, const fraction& f2){
    if (llll(f1) == llll(f2)) return false;
    if (llll(f1) == llll(-1, 0) || llll(f2) == llll(1, 0)) return true;
    if (llll(f1) == llll(1, 0) || llll(f2) == llll(-1, 0)) return false;
    return ld(f1.a) * ld(f2.b) < ld(f2.a) * ld(f1.b);
}

struct line{
    ll a, b;
    line_type t = normal;
    set<line>::iterator* it = new set<line>::iterator;

    line(ll _a, ll _b) : a(_a), b(_b){}
    line(ll _x) : a(0), b(_x), t(value){}
    line(const line& other) : a(other.a), b(other.b), t(other.t){
        *it = *other.it;
    }
    line& operator=(const line& other){
        a = other.a;
        b = other.b;
        t = other.t;
        *it = *other.it;
        return *this;
    }

    ll operator()(ll x) const{
        return a * x + b;
    }

    static line left_edge(){
        line ret(0);
        ret.t = minus_infinity;
        return ret;
    }

    static line right_edge(){
        line ret(0);
        ret.t = infinity;
        return ret;
    }

    ~line(){
        delete it;
    }
};

inline fraction intersection(const line& l1, const line& l2){
    if (l1.t == infinity || l2.t == infinity) return fraction(1, 0);
    if (l1.t == minus_infinity || l2.t == minus_infinity) return fraction(-1, 0);
    return fraction(l1.b - l2.b, l2.a - l1.a);
}

inline bool operator<(const line& l1, const line& l2){
    if (l1.t == normal && l2.t == normal) return llll(-l1.a, l1.b) < llll(-l2.a, l2.b);
    if (l1.t == minus_infinity || l2.t == infinity) return true;
    if (l1.t == infinity || l2.t == minus_infinity) return false;
    if (l1.t == value) return fraction(l1.b) < intersection(*prev(*l2.it), l2);
    return !(fraction(l2.b) < intersection(l1, *next(*l1.it)));
}

struct cht{
    set<line> s = {line::left_edge(), line::right_edge()};

    cht(){
        *(s.begin()->it) = s.begin();
        *(next(s.begin())->it) = next(s.begin());
    }

    void insert_line(const line& l){
        set<line>::iterator it2 = s.lower_bound(l), it1 = prev(it2);
        if (it1 != s.begin() && (it1->a == l.a || !(intersection(*it1, l) < intersection(*it1, *it2)))
            ) return;
        if (it2 != prev(s.end()) && l.a == it2->a) ++it2;
        while (it1 != s.begin() && !(intersection(*prev(it1), *it1) < intersection(*it1, l))) --it1;
        while (it2 != prev(s.end()) && !(intersection(*it2, l) < intersection(*it2, *next(it2)))) ++
            it2;
        while (next(it1) != it2) s.erase(next(it1));
        set<line>::iterator it = s.insert(it1, l);
        *(it->it) = it;
    }

    ll operator()(ll x){
        return (*prev(s.upper_bound(line(x))))(x);
    }
};
```

5.5 Linear Convex Hull

```
struct fraction{
    ll a, b;
    fraction(ll _a, ll _b = 1) : a(_a), b(_b){}
};

bool operator<(const fraction& f1, const fraction& f2){
    return f1.a * f2.b < f2.a * f1.b;
}

struct line{
    ll a, b;

    line(ll _a, ll _b) : a(_a), b(_b){}

    ll operator()(ll x) const{
        return a * x + b;
    }
};

inline fraction intersection(const line& l1, const line& l2){
    return fraction(l1.b - l2.b, l2.a - l1.a);
}

struct cht{
    int pos = 0;
    vector<line> arr = {};

    cht(){}

    void insert_line_at_end(const line& l){
        if (arr.size() != 0 && !(llll(l.a, l.b) < llll(arr.back().a, arr.back().b))) return;
        if (arr.size() != 0 && l.a == arr.back().a) arr.pop_back();
        while (arr.size() >= 2 && !(intersection(arr[arr.size() - 2], arr.back()) < intersection(arr.
            back(), l))) arr.pop_back();
        arr.push_back(l);
    }

    ll operator()(ll x){
        pos = min(pos, int(arr.size()) - 1);
        while (pos != 0 && arr[pos - 1](x) < arr[pos](x)) --pos;
        while (pos != int(arr.size()) - 1 && arr[pos](x) >= arr[pos + 1](x)) ++pos;
        return arr[pos](x);
    }
};
```