Contents

1 Data Structures

1.1 Bit 2d

2D Sum BIT, update and sum. The problem must be 1-indexed.

Query/update time: $\mathcal{O}((\log n)^2)$

Construction time: $\mathcal{O}(n^2(\log n)^2)$

Space: $\mathcal{O}(n^2)$

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
#define MAX 1123

int bit[MAX][MAX], x, y;
void setbit(int i, int j, int delta) {
  int j_;
  while(i <= x) {
    j_ = j;
    while(j_ <= y) {
      bit[i][j_] += delta;
      j_ += j_ & -j_;
    }
    i += i & -i;
  }
}
ll getbit(int i, int j) {
  ll ans = 0;
  int j_;
  while(i) {
    j_ = j;
    while(j_) {
      ans += bit[i][j_];
      j_ -= j_ & -j_;
    }
    i -= i & -i;
  }
  return ans;
}

int main(void) {
  int p;
  while (scanf("%d %d %d", &x, &y, &p), x || y || p) {
    for(int i = 0 ; i <=  x; i++)
      for(int j = 0; j <= y; j++)
        bit[i][j] = 0;
    int q;
    scanf("%d", &q);
    while(q--) {
      char c;
      scanf(" %c",&c);
      int n, xi, yi, zi, wi;
      if(c == 'A') {
        scanf(" %d %d %d", &n, &xi, &yi);
        xi++; yi++;
        setbit(xi, yi, n);
      }
      else {
        scanf(" %d %d %d %d", &xi, &yi, &zi, &wi);
        xi++; yi++; zi++; wi++;
        if(xi > zi) swap(xi, zi);
        if(yi > wi) swap(yi, wi);
        ll ans = getbit(zi, wi) - getbit(zi, yi - 1)
        - getbit(xi - 1, wi) + getbit(xi - 1, yi - 1);
        printf("%lld\n", ans * (ll) p);
      }
    }
    printf("\n");
  }
  return 0;
}
```

1.2 DSU - Disjoint Set Union

Query/update time: $\mathcal{O}(1)$

Construction time: $\mathcal{O}(n)$

Space: $\mathcal{O}(n)$

```
struct DSU {
    vi p, sz;
    DSU(int n) {
        p.resize(n);
        iota(p.begin(), p.end(), 0);
        sz.assign(n, 1);
    }
    int find(int i) {
        if (p[i] == i) return i;
        return p[i] = find(p[i]);
    }
    bool unite(int u, int v) {
        u = find(u);
        v = find(v);
        if (u == v) return false;
        if (sz[u] < sz[v]) swap(u, v);
        p[v] = u;
        sz[u] += sz[v];
        return true;
    }
};
```

1.3 DSU - Binary Tree

Specific code to find maximum path sums between pairs of vertices. Uses Kruskal-style MST. Query/update time: possibly $\mathcal{O}(n)$ Construction time: $\mathcal{O}(n)$ Space: $\mathcal{O}(n)$

```
vi d;
vi_ii e;
vi ans;

int merged;
vi _p, _leaf, _wei;
vvi adj;
int _find(int u) { return _p[u] == u ? u: _p[u] = _find(_p[u]); }
void _union(int u, int v, int w){
  u = _find(u);
  v = _find(v);
  int merge_ind = merged+n;
  _p[u] = merge_ind;
  _p[v] = merge_ind;
  _leaf[merge_ind] = _leaf[u] + _leaf[v];
  _wei[merge_ind] = max(_wei[u], _wei[v]);
  adj[u].push_back(merge_ind);
  adj[merge_ind].push_back(u);
  adj[v].push_back(merge_ind);
  adj[merge_ind].push_back(v);
  merged++;
}
void make(){
  _p = vi(2*n);
  for(int i = 0; i < 2*n; i++) _p[i] = i;
  _leaf = vi(2*n, 1);
  _wei = vi(2*n);
  for(int i = 0; i < n; i++) _wei[i] = d[i];
  merged = 0;
  adj = vvi(2*n);
}

void dfs(int u, int p){
  for(auto &v: adj[u]){
    if(v == p) continue;
    ans[v] = ans[u] + (_leaf[u] - _leaf[v])*_wei[u];
    dfs(v, u);
  }
}
```

1.4 MinQueue

Minimum (or maximum) queue. All operations are $\mathcal{O}(1)$ on average. Useful for fixed length max/min queries

```
struct MinQueue{
  deque<ii> q;
  int added = 0;
  int removed = 0;

  // returns [value,index]
  ii getmin(){ return q.front(); }

  void push(int x){
    while (!q.empty() && q.back().first > x)
      q.pop_back();
    q.push_back({x, added});
    added++;
  }

  void pop(){
    if (!q.empty() && q.front().second == removed)
      q.pop_front();
    removed++;
  }
};
```

1.5 Mo's Algorithm

A technique for solving offline range queries on static arrays by sorting queries to minimize total pointer movement. It processes intervals by incrementally updating the range via `add` and `remove` operations. With the optimal block size, the time complexity is $\mathcal{O}((N+Q)\sqrt{N})$ or $\mathcal{O}(N\sqrt{Q})$, depending on block size choice ($\sqrt{N}$ or $N/\sqrt{Q}$). This example solves queries for distinct elements in range

```
struct Mo {
  struct Query {
    int l, r, idx, b;
    bool operator<(const Query& o) const {
      return b != o.b ? b < o.b :
        (b & 1 ? r > o.r : r < o.r);
    }
  };

  int n, block_sz;

  // custom stuff
  vi freq, a;
  int ans = 0;

  vector<Query> queries;
  Mo(int n) : n(n), block_sz(round(sqrt(n))) {}

  // [l,r] indexed
  void add_query(int l, int r, int i) {
    queries.push_back({l,r,i,l/block_sz});
  }
  void add(int i) {
    // add val at i
    freq[a[i]]++;
    if (freq[a[i]] == 1) ans++;
  }
  void remove(int i) {
    // remove value at i
    freq[a[i]]--;
    if (freq[a[i]] == 0) ans--;
  }
  int get_ans() {
    // compute current answer
    return ans;
  }

  vi run() {
    vi ans(queries.size());
    sort(queries.begin(), queries.end());
    int l = 0, r = -1;
    for (auto& q : queries) {
      while (l > q.l) add(--l);
      while (r < q.r) add(++r);
      while (l < q.l) remove(l++);
      while (r > q.r) remove(r--);
      ans[q.idx] = get_ans();
    }
    return ans;
  }
};
```

1.6 Segment Tree

Segment tree with lazy propagation. Here the interval convention is $[l, r[$, with 0-based indexing. The example solves Kadane (max subarray sum) with point/range updates.

Query/update time: $\mathcal{O}(\log n)$

Construction time: $\mathcal{O}(n)$

Space: $\mathcal{O}(n)$

```
struct segtree {
  struct node {
    int seg, pre, suf, sum;
  };
  int size;
  vector<node> nodes;
  vector<bool> hasLazy;
  vector<int> lazy;

  node NEUTRAL = {0,0,0,0};

  void debug(){
    if (nodes.empty() || size == 0) {
      cout << "[Empty Tree]\n"; return;
    }
```

```cpp

    string indent = "..";
    function<void(int, int, int, string)> print_dfs;

    print_dfs = [&](int x, int lx, int rx, string prefix) {
      cout << prefix << " [" << lx << ", " << rx << ")";

      // debug node
      node a = nodes[x];
      cout << "{ ";
      cout << "seg: " << a.seg << ' ';
      cout << "pre: " << a.pre << ' ';
      cout << "suf: " << a.suf << ' ';
      cout << "sum: " << a.sum << ' ';
      cout << "hasLazy: " << hasLazy[x] << ' ';
      cout << "lazy: " << lazy[x] << ' ';
      cout << "}";
      cout << endl;

      if (rx-lx <= 1) return;

      int mx = (lx+rx)/2;
      print_dfs(2*x+1,lx,mx, prefix + indent);
      print_dfs(2*x+2,mx,rx, prefix + indent);
    };
    print_dfs(0, 0, size, "");
  }

  node single(int v){
    return {v,v,v,v};
  }

  node merge(node a, node b){
    return {
      max(max(a.seg, b.seg), a.suf + b.pre),
      max(a.pre, a.sum + b.pre),
      max(b.suf, b.sum + a.suf),
      a.sum+b.sum
    };
  }

  void init (vi &a){
    int n = a.size();
    size = 1;
    while (size < n) size *= 2;
    nodes.assign(2*size-1, NEUTRAL);
    hasLazy.assign(2*size-1, false);
    lazy.assign(2*size-1, 0);
    build(0,0,size,a);
  }

  void build(int x, int lx, int rx, vi &a){
    if (rx-lx == 1){
      if (lx < a.size()) nodes[x] = single(a[lx]);
      return;
    }
    int mx = (lx+rx)/2;
    build(2*x+1,lx,mx,a);
    build(2*x+2,mx,rx,a);
    nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
  }

  void set(int i, int v, int x, int lx, int rx){
    if (rx-lx == 1){
      nodes[x] = single(v);
      return;
    }
    int mx = (lx+rx)/2;
    if (i < mx) set(i, v, 2*x+1, lx, mx);
    else set(i, v, 2*x+2, mx, rx);
    nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
  }

  void set(int i, int v){
    set(i, v, 0, 0, size);
  }

  void rangeUpdate(int l, int r, int v){
    rangeUpdate(l,r,v,0,0,size);
  }

  void rangeUpdate(int l, int r, int v, int x, int lx, int rx){
    unlazy(x,lx,rx);
    if (rx-lx < 1 || rx <= l || lx >= r) return;
    if (l <= lx && rx <= r) return propagate(x,lx,rx,v);
    int mx = (lx+rx)/2;
    rangeUpdate(l,r,v,2*x+1,lx,mx);
    rangeUpdate(l,r,v,2*x+2,mx,rx);
    nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
  }

  node query(int l, int r){
    return query(l,r,0,0,size);
  }

  node query(int l, int r, int x, int lx, int rx){
    unlazy(x,lx,rx);
    if (rx-lx < 1 || rx <= l || lx >= r) return NEUTRAL;
    if (l <= lx && rx <= r) return nodes[x];
    int mx = (lx+rx)/2;
    node left = query(l,r,2*x+1,lx,mx);
    node right = query(l,r,2*x+2,mx,rx);
    return merge(left,right);
  }

  void unlazy(int x, int lx, int rx){
    if (hasLazy[x]){
      propagate(x,lx,rx,lazy[x]);
      hasLazy[x] = false;
    }
  }

  void propagate(int x, int lx, int rx, int v){
    nodes[x].sum = (rx-lx)*v;
    nodes[x].seg = max((rx-lx)*v,0ll);
    nodes[x].pre = max((rx-lx)*v,0ll);
    nodes[x].suf = max((rx-lx)*v,0ll);
    if (rx-lx > 1){
      lazy[2*x+1] = v;
      lazy[2*x+2] = v;
      hasLazy[2*x+1] = true;
      hasLazy[2*x+2] = true;
    }
  }
};
```

1.7 Sparse Table RMQ

Sparse table for RMQ in $\mathcal{O}(1)$, used in many problems, including $\mathcal{O}(1)$ LCA (Trees) and LCP (SuffixArray) queries.

```cpp

struct SparseTable {
  vector<vector<ii>> st;

  void build(const vi &a) {
    int n = a.size();
    int max_log = __bit_width(n);
    st.assign(max_log, vector<ii>(n));
    for (int i = 0; i < n; i++) {
      st[0][i] = {a[i], i};
    }
    for (int i = 1; i < max_log; i++) {
      for (int j = 0; j + (1 << i) <= n; j++) {
        // Combine the two halves
        st[i][j] = std::min(st[i-1][j], st[i-1][j + (1 << (i-1))]);
      }
    }
  }

  // Returns min value and index in range [l, r] inclusive
  ii min(int l, int r) {
    int len = r - l + 1;
    int k = __bit_width(len) - 1;
    return std::min(st[k][l], st[k][r - (1<<k) + 1]);
  }
};
```

2 Graphs

2.1 BFS 0-1

Time: $\mathcal{O}(n+m)$

```cpp
vi bfs01(int s){
    vi d(n, INF);
    d[s] = 0;
    deque<int> q;
    q.push_front(s);
    while(!q.empty()){
        int u = q.front(); q.pop_front();
        for (auto [w,v] : adj[u]){
            if (d[u]+w < d[v]){
                d[v] = d[u] + w;
                if (w == 1) q.push_back(v);
                else q.push_front(v);
            }
        }
    }
    return d;
}
```

2.2 Bridges and Articulation points

DFS to get bridges and articulation points of a graph in $\mathcal{O}(n+m)$

```cpp
vvi
vi in, low;
int timer;
set<int> cut_points;
vector<ii> bridges;

```

```cpp
void dfs_ap(int u, int p = -1) {
  in[u] = low[u] = ++timer;
  int ch = 0;
  for (int v : adj[u]) {
    if (v == p) continue;
    if (in[v]) {
      // Back-edge
      low[u] = min(low[u], in[v]);
    } else {
      // Tree-edge
      dfs_ap(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] >= in[u] && p != -1)
        cut_points.insert(u);
      ch++;
    }
  }
  if (p == -1 && ch > 1)
    cut_points.insert(u);
}

void dfs_bridges(int u, int p = -1) {
  in[u] = low[u] = ++timer;
  for (int v : adj[u]) {
    if (v == p) continue;
    if (in[v]) {
      low[u] = min(low[u], in[v]);
    } else {
      dfs_bridges(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > in[u])
        bridges.push_back({u, v});
    }
  }
}

void init(int n) {
  timer = 0;
  in.assign(n, 0);
  low.assign(n, 0);
  cut_points.clear();
  bridges.clear();
}
```

2.3 Dijkstra

Time: $\mathcal{O}(m \log n)$

```cpp
void dijkstra(int s){
    int d, u, v;
    dist = vi(n, INF);
    dist[s] = 0;
    priority_queue<ii, vii, greater<ii>> pq;
    pq.emplace(0,s);
    while(!pq.empty()){
        auto [d,u] = pq.top(); pq.pop();
        if (d > dist[u]) continue;

        for (auto &[w,v] : adj[u]){
            if (dist[v] > dist[u] + w){
                dist[v] = dist[u] + w;
                pq.emplace(dist[v], v);
            }
        }
    }
}
```

2.4 Dinic - Flow/matchings

- **General Network:** $\mathcal{O}(VE\log U)$.
- **Unit Capacity Network:** $\mathcal{O}(\min(V^{2/3}, E^{1/2}) \cdot E)$. Often considered $\mathcal{O}(E\sqrt{V})$.
- **Bipartite Matching:** $\mathcal{O}(\min(V^{2/3}, E^{1/2}) \cdot E)$. Often considered $\mathcal{O}(E\sqrt{V})$.

```cpp
struct Dinic {
  struct Edge {
    int u, v;
    ll cap, flow = 0;
    Edge(int u, int v, ll cap) : u(u), v(v), cap(cap)
        {}
  };

  const ll flow_inf = 1e18;
  vector<Edge> edges;
  vvi adj;
  int n, m = 0;
  int s, t;
  vi level, ptr;
  queue<int> q;

  Dinic(int n): n(n) {
    adj.resize(n);
    level.resize(n);
    ptr.resize(n);
  }

  void add_edge(int u, int v, ll cap) {
    edges.emplace_back(u,v,cap);
    edges.emplace_back(v,u,0);
    adj[u].push_back(m++);
    adj[v].push_back(m++);
  }

  bool bfs(ll delta){
    queue<int> q;
    q.push(s);
    while(!q.empty()){
      int u = q.front(); q.pop();
      for (int id : adj[u]){
        auto &e = edges[id];
        if (e.cap - e.flow < delta) continue;
        if (level[e.v] != -1) continue;
        level[e.v] = level[u]+1;
        q.push(e.v);
      }
    }
    return level[t] != -1;
  }

  ll dfs(int u, ll pushed) {
    if (pushed == 0) return 0;
    if (u == t) return pushed;
    for (int &cid = ptr[u]; cid < (int)adj[u].size();
        cid++){
      int id = adj[u][cid];
      auto &e = edges[id];
      if (level[u]+1 != level[e.v]) continue;
      ll tr = dfs(e.v,min(pushed, e.cap - e.flow));
      if (tr == 0) continue;
      e.flow += tr;
      edges[id^1].flow -= tr;
      return tr;
    }
    return 0;
  }

  ll maxflow(int s, int t){
    this->s = s; this->t = t;
    ll max_c = 0;
    for (auto &e : edges) max_c = max(max_c, e.cap);

    ll delta = 1;
    while(delta <= max_c) delta <<= 1;
    delta >>= 1;

    ll f = 0;
    for (;delta > 0; delta >>= 1){
      while(true){
        fill(level.begin(), level.end(),-1);
        level[s] = 0;
        if (!bfs(delta)) break;
        fill(ptr.begin(), ptr.end(), 0);
        while(ll pushed = dfs(s,flow_inf)) f += pushed;
      }
    }
    return f;
  }

  // call constructor with (n1+n2+2) beforehand (dont
      add edges manually)
  // assumes pairs are 1-indexed
  vii maxmatchings(int n1, int n2, const vii& pairs){
    for (int i = 1; i <= n1; i++)
      add_edge(0,i,1);

    for (int i = 1; i <= n2; i++)
      add_edge(i+n1,n-1,1);

    for (auto &[u,v] : pairs)
      add_edge(u,v+n1,1);

    maxflow(0,n-1);

    vii matchings;
    for (auto &e : edges){
      if (e.u >= 1 && e.u <= n1 && e.flow == 1 && e.v >
          n1){
        matchings.emplace_back(e.u,e.v-n1);
      }
    }
    return matchings;
  }


  vii mincut(int s, int t){
    maxflow(s,t);
    queue<int> q; q.push(s);
    vector<bool> reachable(n);
    reachable[s] = true;
    while(!q.empty()){
      int u = q.front(); q.pop();
      for (auto &id : adj[u]){
        int v = edges[id].v;
        if (edges[id].cap - edges[id].flow > 0 && !
            reachable[v]) {
          reachable[v] = true;
          q.push(v);
        }
      }
    }
```

```
    vii minCutEdges;

    for (int i = 0; i < m; i += 2) {
      const Edge& edge = edges[i];
      if (reachable[edge.u] && !reachable[edge.v]) {
        minCutEdges.emplace_back(edge.u, edge.v);
      }
    }

    return minCutEdges;
  }
};
```

2.5 Floyd-Warshall

Time: $\mathcal{O}(n^3)$

```

vvi d(n, vi(n, INF));
void floyd_warshall(){
  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
}
```

2.6 Hopcroft-Karp - Bipartite Matching

Bipartite matching such as Kuhn but faster. BFS until first layer missing match, DFS for the BFS graph to find pairings. Time: $\mathcal{O}(E\sqrt{V})$

```
int n, m, k;
vvi adj;
vi p, dist; /*p is in matching for [0, n[ and parent for [n, n+m[*/

int bfs(){
  queue<int> q;
  dist = vi(n+m, inf);
  for(int i = 0; i < n; i++){
    if(p[i] == -1) q.push(i), dist[i] = 0;
  }
  int min_dist_match = inf;
  while(!q.empty()){
    int u = q.front(); q.pop();
    if(dist[u] > min_dist_match) continue;
    for(auto v: adj[u]){
      if(p[v] == -1) min_dist_match = dist[u];
      else if(dist[p[v]] == inf){
        dist[p[v]] = dist[u] + 1;
        q.push(p[v]);
      }
    }
  }
  return min_dist_match != inf;
}

int dfs(int u){
  for(auto v: adj[u]){
    if(p[v] == -1 || (dist[u]+1 == dist[p[v]] && dfs(p[v]))){
      p[v] = u;
      p[u] = 1;
      return true;
    }
  }
  dist[u] = inf;
  return false;
}

int hopkarp(){
  p = vi(n+m, -1);
  int matchings = 0;
  while(bfs()){
    for(int i = 0; i < n; i++){
      if(p[i] == -1 && dfs(i)) matchings++;
    }
  }
  return matchings;
}

void create(){
  adj = vvi(n+m);
  for(int i = 0; i < k; i++){
    int u, v;
    cin >> u >> v; u--; v--;
    v += n;
    adj[u].push_back(v);
  }
}
```

2.7 Hungarian

Solves minimum cost assignment for n workers and m jobs. Time: $\mathcal{O}((n+m)^3)$

```

// cost should be (cost[worker][job])
pair<int,vii> hungarian(int n, int m, const vvi &cost) {
  if (n == 0) return {0,{}};
  int N = max(n, m);

  vi u(N+1), v(N+1), p(N+1), way(N+1);

  const int INF = 1e9;
  for (int i = 1; i <= n; ++i) {
    p[0] = i;
    int j0 = 0;
    vi minv(N + 1, INF);
    vector<bool> used(N + 1, false);

    do {
      used[j0] = true;
      int i0 = p[j0], delta = INF, j1;

      for (int j = 1; j <= N; ++j) {
        if (!used[j]) {
          int cur = cost[i0-1][j-1] - u[i0] - v[j];
          if (cur < minv[j]) {
            minv[j] = cur;
            way[j] = j0;
          }
          if (minv[j] < delta) {
            delta = minv[j];
            j1 = j;
          }
        }
      }

      for (int j = 0; j <= N; ++j) {
        if (used[j]) {
          u[p[j]] += delta;
          v[j] -= delta;
        } else {
          minv[j] -= delta;
        }
      }
      j0 = j1;
    } while (p[j0] != 0);

    do {
      int j1 = way[j0];
      p[j0] = p[j1];
      j0 = j1;
    } while (j0);
  }

  int total_cost = 0;
  for (int j = 1; j <= m; ++j) {
    if (p[j] != 0) {
      total_cost += cost[p[j] - 1][j - 1];
    }
  }

  // {worker,job}[] 0-indexed
  vii matchings;
  for (int j = 1; j <= m; ++j) {
    if (p[j] != 0) {
      matchings.push_back({p[j] - 1, j - 1});
    }
  }
  return {total_cost, matchings};
}
```

2.8 Kosaraju - SCCs

Computes the strongly connected components of a graph. Also computes the reverse topological order (if it exists). Time: $\mathcal{O}(n+m)$

```
void dfs1(int u){
    vis[u] = 1;
    for (auto v : adj[u]){
        if (!vis[v]) dfs1(v);
    }

    ts.push_back(u);
}

void dfs2(int u, int c){
    scc[u] = c;
    for (auto v : adjT[u])
        if (!scc[v]) dfs2(v,c);
}


// usage
for (int i = 0; i < n; i++)
    if (!vis[i]) dfs1(i);

reverse(ts.begin(), ts.end());

int c = 1;
for (auto u : ts)
    if (!scc[u]) dfs2(u,c++);
```

2.9 Kuhn - Bipartite Matching

Bipartite matching. Time: $\mathcal{O}(VE)$

```cpp
int matchings;
vi p, vis;
vii match;

int dfs(int u){
  if(vis[u]) return 0;
  vis[u] = 1;
  for(auto v: adj[u]){
    if(p[v] == -1 || dfs(p[v])){
      p[v] = u;
      return 1;
    }
  }
  return 0;
}

void kuhn(){
  matchings = 0;
  p = vi(n+m, -1);
  for(int i = 0; i < n; i++){
    vis = vi(n, 0);
    matchings += dfs(i);
  }
  for(int i = n; i < n+m; i++){
    if(p[i] != -1) match.push_back(ii(p[i], i));
  }
}

void create(){
  adj = vvi(n+m);
  for(int i = 0; i < k; i++){
    int u, v;
    cin >> u >> v; u--; v--;
    adj[u].push_back(v+n);
  }
}
```

2.10 Min cost flow

Time: $\mathcal{O}(FE \log V)$

If negative costs are needed (maximize cost), need to run SPFA once at the start, making the solution $\mathcal{O}(EV + FE \log V)$.

```cpp
struct MinCostFlow {
  struct Edge {
    int to, capacity, rev;
    ll cost;
  };

  int n;
  vector<vector<Edge>> adj;

  MinCostFlow(int _n) : n(_n), adj(_n) {}

  void add_edge(int from, int to, int cap, ll cost){
    adj[from].push_back({to,cap,(int)adj[to].size(),
        cost});
    adj[to].push_back({from,0,(int)adj[from].size()-1,
        -cost});
  }

  // O(FElog(V))
  lli min_cost_flow(int s, int t, int targetFlow) {
    int flow = 0;
    ll total_cost = 0;
    vll dist, h(n);
    vi pv, pe;

    // needed only if negative costs exists
    spfa(s, h, pv, pe);

    while (flow < targetFlow) {
      dijkstra(s, h, dist, pv, pe);

      if (dist[t] == INF) break;

      for (int i = 0; i < n; i++) {
        if (dist[i] < INF) {
          h[i] += dist[i];
        }
      }

      int f = targetFlow - flow;
      int cur = t;
      while (cur != s) {
        f = min(f, adj[pv[cur]][pe[cur]].capacity);
        cur = pv[cur];
      }

      flow += f;
      total_cost += f * h[t];
      cur = t;
      while (cur != s) {
        Edge &e = adj[pv[cur]][pe[cur]];
        e.capacity -= f;
        adj[e.to][e.rev].capacity += f;
        cur = pv[cur];
      }
    }


    return {total_cost, flow};
  }

  // needed only if negative costs exists
  void spfa(int s, vll &dist, vi &pv, vi &pe) {
    dist.assign(n, INF);
    pv.assign(n,-1);
    pe.assign(n,-1);
    vector<bool> inq(n, false);
    queue<int> q;

    dist[s] = 0;
    q.push(s);
    inq[s] = true;

    while (!q.empty()) {
      int u = q.front(); q.pop();
      inq[u] = false;
      for (int i = 0; i < adj[u].size(); i++) {
        Edge &e = adj[u][i];
        int v = e.to;
        if (e.capacity > 0 && dist[v] > dist[u] + e.
            cost) {
          dist[v] = dist[u] + e.cost;
          pv[v] = u;
          pe[v] = i;
          if (!inq[v]) {
            inq[v] = true;
            q.push(v);
          }
        }
      }
    }
  }

  void dijkstra(int s, vll &h, vll &dist, vi &pv, vi &
      pe) {
    dist.assign(n, INF);
    pv.assign(n, -1);
    pe.assign(n, -1);
    dist[s] = 0;

    priority_queue<lli, vector<lli>, greater<lli>> pq;
    pq.emplace(0, s);

    while (!pq.empty()) {
      auto [d, u] = pq.top(); pq.pop();
      if (d > dist[u]) continue;

      for (int i = 0; i < adj[u].size(); i++) {
        Edge &e = adj[u][i];
        if (e.capacity <= 0) continue;
        int v = e.to;

        ll reduced_cost = e.cost + h[u] - h[v];
        if (dist[u] != INF && dist[v] > dist[u] +
            reduced_cost) {
          dist[v] = dist[u] + reduced_cost;
          pv[v] = u;
          pe[v] = i;
          pq.push({dist[v], v});
        }
      }
    }
  }
};

// usage
int nodes = 302; // amount of nodes in the network
MinCostFlow mcf(nodes);

for (int i = 0; i < 150; i++){
  mcf.add_edge(0, i+1, 1, 0); // source to node
  mcf.add_edge(i+151, nodes-1, 1, 0); // node to sink
}

for (int i = 0; i < n; i++){
  int a, b, c; cin >> a >> b >> c;
  mcf.add_edge(a, b+150, 1, -c); // edges in between (-
      c to maximize the cost)
}

// final max cost is -cost
auto [cost, flow] = mcf.min_cost_flow(0,nodes-1,150);
```

2.11 MST - Kruskal

Time: $\mathcal{O}(m \log m)$

```cpp
vector<pair<int,ii>> edges; // [weight, (u,v)]
int kruskal(int n){
    int cost = 0;
    DSU dsu(n); // n is the numb of vertices
    sort(edges.begin(), edges.end());
    for (auto &[w,uv] : edges){
        auto [u,v] = uv;
        if (dsu.unite(u,v)) cost += w;
```

```cpp
    }
    return cost;
}
```

2.12 MST - Prim

Time: $\mathcal{O}(m \log n)$

```cpp
vvii adj, mst;
vi taken;

int prim(){
    priority_queue<iii, vector<iii>, greater<iii>> pq;
    taken[0] = 1;
    for (auto [w,v] : adj[0]){
        if (!taken[v]) pq.push({w, {0,v}});
    }

    int cost = 0;
    while (!pq.empty()){
        auto [w,pu] = pq.top(); pq.pop();
        auto [p,u] = pu;
        if (!taken[u]) {
            cost += w;
            mst[p].emplace_back(w,u);
            mst[u].emplace_back(w,p);
            taken[u] = 1;
            for (auto [w,v] : adj[u]){
                if (!taken[v]) {
                    pq.push({w,{u,v}});
                }
            }
        }
    }
    return cost;
}
```

2.13 SCC compressing

Condensing a graph into a DAG through its strongly connected components can be useful for DP

```cpp
struct SCCCondenser {
  int n, timer, scc_cnt;
  vi in, low, scc;
  stack<int> st;

  SCCCondenser(const vvi& adj) {
    n = adj.size();
    in.assign(n, 0);
    low.assign(n, 0);
    scc.assign(n, -1);
    timer = scc_cnt = 0;
    for (int i = 0; i < n; ++i)
      if (!in[i]) dfs(i, adj);
  }

  void dfs(int u, const vvi& adj) {
    in[u] = low[u] = ++timer;
    st.push(u);
    for (int v : adj[u]) {
      if (!in[v]) {
        dfs(v, adj);
        low[u] = min(low[u], low[v]);
      } else if (scc[v] == -1) {
        low[u] = min(low[u], in[v]);
      }
    }
    if (low[u] == in[u]) {
      while (true) {
        int v = st.top(); st.pop();
        scc[v] = scc_cnt;
        if (u == v) break;
      }
      scc_cnt++;
    }
  }

  // Returns {DAG, Aggregated Values}
  pair<vvi, vi> build(const vvi& adj, const vi& val) {
    vvi dag(scc_cnt);
    vi scc_val(scc_cnt);
    set<ii> edges;

    for (int u = 0; u < n; ++u) {
      scc_val[scc[u]] += val[u]; // Aggregate values
      for (int v : adj[u]) {
        if (scc[u] != scc[v]) {
          if (edges.count({scc[u], scc[v]})) continue;
          edges.insert({scc[u], scc[v]});
          dag[scc[u]].push_back(scc[v]);
        }
      }
    }
    return {dag, scc_val};
  }
};
```

3 DP

3.1 Bin Packing

Time: $\mathcal{O}(n \cdot 2^n)$ Space: $\mathcal{O}(2^n)$

```cpp
vi w(n);

vector<ii> dp(1<<n, ii(INF,0));
// dp[i] = for the subset i(bitmask) (A,B) is the pair where
// A - the min number of knapsacks to store this subset
// B - the min size of a used knapsack

dp[0] = ii(0,INF);
for (int subset = 1; subset < (1<<n); subset++){
  for (int item = 0; item < n; item++){
    if (!((subset>>item)&1)) continue;
    int prevsubset = subset - (1<<item);
    ii prev = dp[prevsubset];

    if (prev.second + w[item] <= x) {
      // can fill the knapsack, fill it
      dp[subset] = min(dp[subset], ii(prev.first, prev.second+w[item]));
    } else {
      // cant fill the knapsack, create a new one
      dp[subset] = min(dp[subset], ii(prev.first+1, w[item]));
    }
  }
}

cout << dp[(1<<n)-1].first << endl;
```

3.2 Broken Profile DP

Solves the problem of counting how many ways to fill an nxm grid using 1x2 tiles. This technique can be used whenever the state dependence is only on the previous state (column). Time: $\mathcal{O}(mn2^n)$ Space: $\mathcal{O}(mn2^n)$

```cpp
int dp[1002][12][1024];
dp[0][0][0] = 1;

for (int i = 0; i < m; i++){
  for (int j = 0; j < n; j++){
    for (int mask = 0; mask < (1<<n); mask++){
      if (mask & (1<<j)){
        int nxt_mask = mask - (1<<j);
        dp[i][j+1][nxt_mask] += dp[i][j][mask];
        dp[i][j+1][nxt_mask] %= M;
      } else {
        int q = mask + (1 << j);
        dp[i][j+1][q] += dp[i][j][mask];
        dp[i][j+1][q] %= M;
        if (j < n-1 && (mask & (1<<(j+1)))==0){
          q = mask + (1 << (j+1));
          dp[i][j+1][q] += dp[i][j][mask];
          dp[i][j+1][q] %= M;
        }
      }
    }
  }

  for (int p = 0; p < (1<<n); p++){
    dp[i+1][0][p] = dp[i][n][p];
  }
}
```

3.3 Convex Hull Trick (CHT)

- **Recurrence form:**

TODO formulas

- **Slope monotonicity:** If coefficients a_j (slopes) are inserted in strictly decreasing (or increasing) order as j grows, and
- **Query monotonicity:** Values x_i for query come in non-decreasing (min) or increasing (max) order consistent with slope order,
- **Complexity:**
  - Insertion + amortized query in $\mathcal{O}(1)$ per operation (pointer walk) under monotonicity.
  - Non-monotonic case, generic CHT via binary search: $\mathcal{O}(\log n)$ per query.
  - General alternative: Li Chao Tree for insertions/queries in arbitrary order, $\mathcal{O}(\log M)$ per operation (where M is the domain of x).

- **Constraints:**
  - If it cannot be written in linear form, CHT does not apply.
  - If there is no monotonicity of slopes or queries, consider Li Chao Tree or CHT variant with binary search.

The example below solves the *dp* where the recurrence is:

TODO formulas

```
struct CHT {
  struct Line { // y = mx + c
    int m, c;
    Line(int m, int c) : m(m), c(c) {}
    int val(int x){
      return m*x + c;
    }
    int floorDiv(int num, int den) {
      if (den < 0) num = -num, den = -den;
      if (num >= 0) return num / den;
      else return - ( (-num + den - 1) / den );
    }
    int ceilDiv(int num, int den) {
      if (den < 0) num = -num, den = -den;
      if (num >= 0) return (num + den - 1) / den;
      else return - ( (-num) / den );
    }
    int intersect(Line l){
      // m1x + c1 = m2x + c2
      // x = (c2 - c1)/(m1 - m2)
      // if slopes are increasing, use floor div
      return ceilDiv(l.c - c, m - l.m);
    }
  };

  deque<pair<Line, int>> dq;

  void insert(int m, int c){
    Line newLine(m, c);
    if (!dq.empty() && newLine.m == dq.back().first.m) {
      // If slopes increasing, change to <=
      if (newLine.c >= dq.back().first.c) return;
      else dq.pop_back();
    }
    // if slopes increasing, change to <=
    while (dq.size() > 1 && dq.back().second >= dq.back().first.intersect(newLine)){
      dq.pop_back();
    }
    if (dq.empty()){
      // assuming queries are positive numbers, may change to -INF or +INF if needed
      dq.emplace_back(newLine, 0);
      return;
    }
    dq.emplace_back(newLine, dq.back().first.intersect(newLine));
  }

  // dont use query and queryNonMonotonicValues in the same problem
  int query(int x){
    while (dq.size() > 1){
      // if slopes increasing, change to >=
      if (dq[1].second <= x) dq.pop_front();
      else break;
    }
    return dq[0].first.val(x);
  }


  int queryNonMonotonicValues(int x){
    int l=0, r=dq.size()-1, ans=0;
    while (l <= r) {
      int mid = (l+r)>>1;
      if (dq[mid].second <= x) {
        ans = mid;
        l = mid + 1;
      } else {
        r = mid - 1;
      }
    }
    return dq[ans].first.val(x);
  }
};

void solve(){
  int n, c; cin >> n >> c;
  vi h(n);
  for (auto &x : h) cin >> x;

  vi dp(n);
  dp[0] = 0;
  CHT cht;
  cht.insert(-2*h[0], h[0]*h[0]);
  for (int i = 1; i < n; i++){
    dp[i] = cht.query(h[i]) + c + h[i]*h[i];
    cht.insert(-2*h[i], h[i]*h[i] + dp[i]);
  }
  cout << dp[n-1] << endl;
}
```

3.4 Edit Distance (Levenshtein)

Very similar to LCS, in the sense that it considers prefixes already computed. Time: $\mathcal{O}(mn)$ Space: $\mathcal{O}(mn)$

```
vvi dp(n+1, vi(m+1));
for (int i = 0; i <= n; i++) dp[i][0] = i;
for (int i = 0; i <= m; i++) dp[0][i] = i;
for (int i = 1; i <= n; i++){
    for (int j = 1; j <= m; j++){
        dp[i][j] = min(
            min(dp[i][j-1]+1, dp[i-1][j]+1),
            dp[i-1][j-1]+(s[i-1]!=t[j-1])
        );
    }
}
```

3.5 Knapsack - 1D

The spirit here is the same as the 2D version, but here it iterates on the knapsack capacity backwards, to ensure that the value of `dp[j-w[i]]` is not considering the item `i`. Time: $\mathcal{O}(nW)$ Space: $\mathcal{O}(W)$

```
vi dp(W+1);
for (int i = 0; i < n; i++){
  for (int j = W; j >= w[i]; j--){
    dp[j] = max(dp[j], v[i] + dp[j-w[i]]);
  }
}
```

3.6 Knapsack - 2D

Time: $\mathcal{O}(nW)$ Space: $\mathcal{O}(nW)$

```
vvi dp(n+1,vi(W+1));
for (int c = 1; c <= W; c++){
  for (int i = 1; i <= n; i++){
    dp[i][c] = dp[i-1][w];
    if (c-w[i-1] >= 0) {
      dp[i][c] = max(dp[i][c], dp[i-1][c-w[i-1]] + v[i-1]);
    }
  }
}
```

3.7 LCS - Longest Common Subsequence

Subsequence generation included here. Time: $\mathcal{O}(mn)$ Space: $\mathcal{O}(mn)$

```
vvi dp(n+1,vi(m+1));
vvii p(n+1,vii(m+1));

for (int i = 1; i <= n; i++){
  for (int j = 1; j <= m; j++){
    if (a[i-1] == b[j-1]) {
      dp[i][j] = dp[i-1][j-1]+1;
      p[i][j] = {i-1,j-1};
    } else if (dp[i][j-1] > dp[i-1][j]){
      dp[i][j] = dp[i][j-1];
      p[i][j] = {i,j-1};
    } else {
      dp[i][j] = dp[i-1][j];
      p[i][j] = {i-1,j};
    }
  }
}

ii pos = ii(n,m);
stack<int> st;
while(pos != ii(0,0)){
  auto [i,j] = pos;
  if (p[i][j] == ii(i-1,j-1)) st.push(a[i-1]);
  pos = p[i][j];
}
cout << st.size() << endl;
while (!st.empty()) {
  cout << st.top() << ' ';
  st.pop();
}
cout << endl;
```

3.8 LiChao Tree

Generalization of CHT for linear functions that do not need to be sorted. Inspired by segtree. Queries and insertions are all $\mathcal{O}(\log M)$. Where M is the size of the query interval the tree receives.

```cpp
// Li Chao tree for minimum (or maximum) over domain [L, R].
// T should support +, *, comparisons.
// For integer x use eps = 0 and discrete mid+1 splitting;
// For floating use eps > 0 and continuous splitting without +1.
template<typename T>
struct lichao_tree {
  // if max lichao, change to ::min()
  static const T identity = numeric_limits<T>::max();

  struct Line {
    T m, c;
    Line() {
      m = 0;
      c = identity;
    }
    Line(T m, T c) : m(m), c(c) {}
    T val(T x) { return m * x + c; }
  };

  struct Node {
    Line line;
    Node *lc, *rc;
    Node() : lc(0), rc(0) {}
  };

  T L, R, eps;
  deque<Node> buffer;
  Node* root;

  Node* new_node() {
    buffer.emplace_back();
    return &buffer.back();
  }

  lichao_tree() {}

  lichao_tree(T _L, T _R, T _eps) {
    init(_L, _R, _eps);
  }

  void clear() {
    buffer.clear();
    root = nullptr;
  }

  void init(T _L, T _R, T _eps) {
    clear();
    L = _L;
    R = _R;
    eps = _eps;
    root = new_node();
  }

  void insert(Node* &cur, T l, T r, Line line) {
    if (!cur) {
      cur = new_node();
      cur->line = line;
      return;
    }

    T mid = l + (r - l) / 2;
    if (r - l <= eps) return;

    // if max lichao, change to >
    if (line.val(mid) < cur->line.val(mid))
      swap(line, cur->line);

    // if max lichao, change to >
    if (line.val(l) < cur->line.val(l)) insert(cur->lc, l, mid, line);
    else insert(cur->rc, mid + 1, r, line);
  }

  T query(Node* &cur, T l, T r, T x) {
    if (!cur) return identity;

    T mid = l + (r - l) / 2;
    T res = cur->line.val(x);
    if (r - l <= eps) return res;

    // if max lichao, change min to max
    if (x <= mid) return min(res, query(cur->lc, l, mid, x));
    else return min(res, query(cur->rc, mid + 1, r, x));
  }

  void insert(T m, T c) { insert(root, L, R, Line(m, c)); }

  T query(T x) { return query(root, L, R, x); }
};
```

3.9 LIS - Longest Increasing Subsequence

Time: $\mathcal{O}(n \log n)$

```cpp
int lis(vi &a){
    int n = a.size();
    vi len(n+1, INF);
    len[0] = -INF;
    for (int i = 0; i < n; i++){
        int l = upper_bound(len.begin(), len.end(), a[i]) - len.begin();
        if(len[l-1] < a[i] && a[i] < len[l]) len[l] = a[i];
    }

    int ans = 0;
    for (int i = 0; i <= n; i++){
        if (len[i] < INF) ans = i;
    }
    return ans;
}
```

3.10 SOSDP

```cpp
int k; // amount of bits
vi a(1<<k);
// sosdp
for (int bit = 0; bit < k; bit++){
  for (int mask = 0; mask < (1<<k); mask++){
    if ((1<<bit) & mask) {
      a[mask] += a[mask ^ (1<<bit)];
    }
  }
}

// do stuff (such as multiplication for OR convolution)

// sosdp inverse
for (int bit = 0; bit < k; bit++){
  for (int mask = 0; mask < (1<<k); mask++){
    if ((1<<bit) & mask) {
      a[mask] -= a[mask ^ (1<<bit)];
    }
  }
}
```

3.11 Subset Sum

Almost identical to Knapsack, this code contains the subset reconstruction. Time: $\mathcal{O}(nS)$ Space: $\mathcal{O}(nS)$

```cpp
vvi dp(n+1,vi(sum+1));
vvii p(n+1,vii(sum+1));

dp[0][0] = 1;

for (int i = 1; i <= n; i++){
  for (int s = 1; s <= sum; s++){
    if (s-a[i-1] >= 0 && dp[i-1][s-a[i-1]]){
      // sum is possible taking item i
      p[i][s] = {i-1,s-a[i-1]};
      dp[i][s] = 1;
    } else if (dp[i-1][s]) {
      // sum not possible taking item i
      // but still possible with other items (<i)
      p[i][s] = {i-1,s};
      dp[i][s] = 1;
    }
  }
}

if (!dp[n][target]) {
  cout << -1 << endl;
  return;
}

vi subset;
ii pos = {n,target};
while(pos != ii(0,0)){
  auto [i,s] = pos;
  if (p[i][s].second != s) subset.push_back(a[i-1]);
  pos = p[i][s];
}
```

4 Trees

4.1 Sum of distances

Given a tree, $f(u,v) :=$ distance from u to v in the tree, compute

$$\sum_{u,v} f(u,v)$$

. Time: $\mathcal{O}(n)$

```cpp
vvi adj;
vi sum_going_down, sum_going_up, sz;

```

```cpp
void dfs(int u, int p){
    for (auto v : adj[u]){
        if (v == p) continue;
        dfs(v,u);
        sz[u] += sz[v];
        sum_going_down[u] += sum_going_down[v];
    }
    sum_going_down[u] += sz[u];
}

void dfs2(int u, int p, int par_ans){
    int up_amount = sz[0] - sz[u];
    sum_going_up[u] += par_ans + up_amount;
    int sum = sum_going_down[u];
    for (auto v : adj[u]){
        if (v == p) continue;
        int par_amount = sz[0] - sz[v];
        dfs2(v,u, par_ans + par_amount + sum - (sum_going_down[v]+sz[v]));
    }
}

void solve(){
    int n; cin >> n;
    adj = vvi(n);
    sum_going_down = sum_going_up = vi(n);
    sz = vi(n,1);

    for (int i = 1; i < n; i++){
        int a, b; cin >> a >> b;
        a--; b--;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    dfs(0,0);
    dfs2(0,0,0);

    for (int i = 0; i < n; i++){
        cout << sum_going_down[i]+sum_going_up[i] << ' ';
    }
    cout << endl;
}
```

4.2 Edge HLD

Sometimes the value is on the edges, for this few things need to change, but here is a template. Pre-computation: $\mathcal{O}(n)$ Queries: $\mathcal{O}(\log^2 n)$

```cpp

struct EdgeHLD {
  int n, timer = 0;
  vvii adj;
  vi parent, depth, size, heavy, head, value;
  vi tin, tout;

  segtree seg;

  void init(int _n, vvii& _adj) {
    n = _n;
    adj = _adj;
    value.assign(n, 0);
    parent.assign(n, -1);
    depth.assign(n, 0);
    size.assign(n, 0);
    heavy.assign(n, -1);
    head.assign(n, 0);
    tin.assign(n, 0);
    tout.assign(n, 0);
    timer = 0;

    // edgeWeight[v] = weight of edge (parent[v], v), for v>0
    // root (0) has no parent, so its value is dummy (0)

    dfs1(0,0,0);
    dfs2(0, 0);

    vi linear(n);
    for (int u = 0; u < n; u++)
      linear[tin[u]] = value[u]; // position stores edge weight

    seg.init(linear);
  }

  int dfs1(int u, int p, int w) {
    size[u] = 1;
    parent[u] = p;
    value[u] = w;
    int max_sz = 0;
    for (auto[v,w] : adj[u]) {
      if (v == p) continue;
      depth[v] = depth[u] + 1;
      int sz = dfs1(v, u, w);
      size[u] += sz;
      if (sz > max_sz) {
        max_sz = sz;
        heavy[u] = v;
      }
    }
    return size[u];
  }

  void dfs2(int u, int h) {
    tin[u] = timer++;
    head[u] = h;
    if (heavy[u] != -1)
      dfs2(heavy[u], h);
    for (auto [v,w] : adj[u]) {
      if (v != parent[u] && v != heavy[u])
        dfs2(v, v);
    }
    tout[u] = timer;
  }

  // u deve ser o filho
  void update_edge(int u, int val) {
    seg.set(tin[u], val);
  }

  void rangeUpdate(int u, int v, int x) {
    while (head[u] != head[v]) {
      if (depth[head[u]] < depth[head[v]]) swap(u, v);
      seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
      u = parent[head[u]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    seg.rangeUpdate(tin[u] + 1, tin[v] + 1, x); // +1 to skip LCA's edge
  }

  void update_subtree(int u, int x) {
    // updates all edges in subtree of u (skip incoming edge to u)
    seg.rangeUpdate(tin[u] + 1, tout[u], x);
  }

  segtree::node query(int u, int v) {
    segtree::node res = seg.NEUTRAL;
    while (head[u] != head[v]) {
      if (depth[head[u]] < depth[head[v]]) swap(u, v);
      res = seg.merge(res, seg.query(tin[head[u]], tin[u] + 1));
      u = parent[head[u]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    res = seg.merge(res, seg.query(tin[u] + 1, tin[v] + 1)); // skip LCA's edge
    return res;
  }

  segtree::node query_subtree(int u) {
    // query all edges in subtree of u
    return seg.query(tin[u] + 1, tout[u]);
  }
};
```

4.3 HLD - Heavy light decomposition

If you need to compute a function on a path in a tree and need to support value updates on nodes, HLD is the way. Pre-computation: $\mathcal{O}(n)$ Queries: $\mathcal{O}(\log^2 n)$

OBS: this implementation uses the same segtree as this notebook, with 0-indexing and open-closed interval convention. Ideally, just change the segtree to change the computed function, the HLD struct remains the same. OBS2: this template also supports mass updates (path/subtree) and subtree queries.

```cpp
struct HLD {
  int n, timer = 0;
  vvi adj;
  vi parent, depth, size, heavy, head, value;
  vi tin, tout;

  segtree seg;

  void init(int _n, vi& _value, vvi& _adj) {
    n = _n;
    adj = _adj;
    value = _value;
    parent.assign(n, -1);
    depth.assign(n, 0);
    size.assign(n, 0);
    heavy.assign(n, -1);
    head.assign(n, 0);
    tin.assign(n, 0);
    tout.assign(n, 0);
    timer = 0;

    dfs1(0);
    dfs2(0, 0);

    vi linear(n);
    for (int u = 0; u < n; u++)
      linear[tin[u]] = value[u];
```

```cpp

    seg.init(linear);
  }

  int dfs1(int u) {
    size[u] = 1;
    int max_sz = 0;
    for (int v : adj[u]) {
      if (v == parent[u]) continue;
      parent[v] = u;
      depth[v] = depth[u] + 1;
      int sz = dfs1(v);
      size[u] += sz;
      if (sz > max_sz) {
        max_sz = sz;
        heavy[u] = v;
      }
    }
    return size[u];
  }

  void dfs2(int u, int h) {
    tin[u] = timer++;
    head[u] = h;
    if (heavy[u] != -1)
      dfs2(heavy[u], h);
    for (int v : adj[u]) {
      if (v != parent[u] && v != heavy[u])
        dfs2(v, v);
    }
    tout[u] = timer;
  }

  void update(int u, int val) {
    seg.set(tin[u], val);
  }

  void rangeUpdate(int u, int v, int x) {
    while (head[u] != head[v]) {
      if (depth[head[u]] < depth[head[v]]) swap(u, v);
      seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
      u = parent[head[u]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    seg.rangeUpdate(tin[u], tin[v] + 1, x);
  }

  void update_subtree(int u, int x) {
    seg.rangeUpdate(tin[u], tout[u], x);
  }

  segtree::node query(int u, int v) {
    segtree::node res = seg.NEUTRAL;
    while (head[u] != head[v]) {
      if (depth[head[u]] < depth[head[v]])
        swap(u, v);
      res = seg.merge(res, seg.query(tin[head[u]], tin[u]+1));
      u = parent[head[u]];
    }
    if (depth[u] > depth[v]) swap(u, v);
    res = seg.merge(res, seg.query(tin[u], tin[v]+1));
    return res;
  }

  segtree::node query_subtree(int u){
    return seg.query(tin[u], tout[u]);
  }
};
```

4.4 LCA - RMQ

Pre-computation: $\mathcal{O}(n \log n)$

Queries: $\mathcal{O}(1)$

```cpp
vvi ch;
vi tin, dep, et_nodes, et_depths;
int timer = 0;
int n;

SparseTable st; // same as in this handbook

void dfs(int u) {
  et_nodes.push_back(u);
  et_depths.push_back(dep[u]);
  tin[u] = timer++;

  for (int v : ch[u]) {
      dep[v] = dep[u] + 1;
      dfs(v);
      et_nodes.push_back(u);
      et_depths.push_back(dep[u]);
  }

  timer++;
}

int lca(int u, int v) {
  int tu = tin[u];
  int tv = tin[v];
  if (tu > tv) swap(tu, tv);
  auto [val,id] = st.min(tu,tv);
  return et_nodes[id];
}

// pre allocation and dfs call
ch = vvi(n);
tin = vi(n);
dep = vi(n);
et_nodes.reserve(2 * n);
et_depths.reserve(2 * n);

dfs(0);
st.build(et_depths);
```

4.5 LCA - binary lifting

Pre-computation: $\mathcal{O}(n \log n)$ Queries: $\mathcal{O}(\log n)$ OBS: just call `dfs(root)` before starting queries.

```cpp
vvi adj, up;
vi tin, tout;
int timer = 0;

void dfs(int u, int p){
    tin[u] = timer++;
    for (auto v : adj[u]){
        if (v == p) continue;
        up[v][0] = u;
        for (int dist = 1; dist < LOGN; dist++){
            up[v][dist] = up[up[v][dist-1]][dist-1];
        }
        dfs(v);
    }
    tout[u] = timer++;
}

int isAncestor(int u, int v){
    return tin[u] <= tin[v] && tout[v] <= tout[u];
}

int lca(int u, int v){
    if (isAncestor(u,v)) return u;
    if (isAncestor(v,u)) return v;
    for (int dist = LOGN-1; dist >= 0; dist--){
        if(!isAncestor(up[u][dist],v)) u = up[u][dist];
    }
    return up[u][0];
}
```

5 Problemas clássicos

5.1 2SAT

Struct for solving 2SAT problems that supports many types of boolean expressions. To add a negated literal use ~u

```cpp
// para adicionar negacao usar ~u
// Ex: a clausula (a v !b) se traduz para add_or(a,~b)
struct TwoSatSolver {
    int n;
    vvi adj, adjT;
    vector<bool> vis, assignment;
    vi topo, scc;

    void build(int _n){
        n = 2*_n;
        adj.assign(n,vi());
        adjT.assign(n,vi());
    }

    int get(int u){
        if (u < 0) return 2*(~u)+1;
        else return 2*u;
    }

    // u -> v
    void add_impl(int u, int v){
        u = get(u), v = get(v);
        adj[u].push_back(v);
        adjT[v].push_back(u);
        adj[v^1].push_back(u^1);
        adjT[u^1].push_back(v^1);
    }

    // u || v
    void add_or(int u, int v){
        add_impl(~u, v);
    }

    // u && v
    void add_and(int u, int v){
        add_or(u,u); add_or(v,v);
    }

    // u ^ v (equiv of x != v)
    void add_xor(int u, int v){
        add_impl(u, ~v);
        add_impl(~u, v);
    }

    // u == v
```

```cpp
    void add_equals(int u, int v){
        add_impl(u, v);
        add_impl(v, u);
    }

    void toposort(int u){
        vis[u] = true;
        for (int v : adj[u])
            if (!vis[v]) toposort(v);
        topo.push_back(u);
    }

    void dfs(int u, int c){
        scc[u] = c;
        for (int v : adjT[u])
            if (!scc[v]) dfs(v,c);
    }

    pair<bool, vector<bool>> solve(){
        topo.clear();
        vis.assign(n, false);

        for (int i = 0; i < n; i++)
            if (!vis[i]) toposort(i);

        reverse(topo.begin(), topo.end());

        scc.assign(n, 0);
        int c = 0;
        for (int u : topo)
            if (!scc[u]) dfs(u,++c);

        assignment.assign(n/2, false);
        for (int i = 0; i < n; i += 2){
            if (scc[i] == scc[i+1]) return {false, {}};
            assignment[i/2] = scc[i] > scc[i+1];
        }

        return {true, assignment};
    }
};
```

5.2 Next Greater Element

One of the classic stack applications. Easy to translate to lower, leq or geq, just change the comparator of the `while`.

```cpp
vi next_greater_elem(n, n);

stack<ii> st;
for (int i = 0; i < n; i++){
    while (!st.empty() && st.top().first < h[i]){
        next_greater_elem[st.top().second] = i;
        st.pop();
    }
    st.emplace(h[i], i);
}
```

6 Strings

6.1 Hashing

Creation time: $\mathcal{O}(n)$ Access time: $\mathcal{O}(1)$ Space: $\mathcal{O}(n)$

```cpp
class Hashing{
    const int mod0 = 1e9+7;
    vi pmod0;
    vull pmod1;

    public:
    void CalcP(int mn, int n){
        random_device rd;
        uniform_int_distribution<int> dist(mn+2, mod0
            -1);
        int p = dist(rd);
        if(p % 2 == 0) p--;
        pmod0 = vi(n);
        pmod1 = vull(n);
        pmod0[0] = pmod1[0] = 1;
        for(int i = 1; i < n; i++){
            pmod0[i] = (pmod0[i-1] * p) % mod0;
            pmod1[i] = (pmod1[i-1] * p);
        }
    }

    viull DistincSubstrHashes(string base, int
         offsetVal){
        int n = base.size();
        viull ans;
        for(int i = 0; i < n; i++){
            int h0 = 0;
            ull h1 = 0;
            for(int j = i; j < n; j++){
                h0 = (h0 + (base[j]-offsetVal)*pmod0[j-
                    i]) % mod0;
                h1 = (h1 + (base[j]-offsetVal)*pmod1[j-
                    i]);
                ans.push_back(iull(h0, h1));
            }
        }
        sort(ans.begin(), ans.end());
        auto last = unique(ans.begin(), ans.end());
        ans.erase(last, ans.end());
        return ans;
    };

    viull WindowHash(string data, int offsetVal, int
         lenWindow){
        int n = data.size();
        int h0 = 0;
        ull h1 = 0;
        viull ans;
        for(int i = 0; i < lenWindow; i++){
            h0 = (h0 + (data[i]+offsetVal)*pmod0[i]) %
                mod0;
            h1 = (h1 + (data[i]+offsetVal)*pmod1[i]);
        }
        ans.push_back(iull((h0*pmod0[n-1])%mod0, h1*
         pmod1[n-1]));
        for(int i = lenWindow; i < n; i++){
            h0 = (h0 - (data[i-lenWindow]+offsetVal)*
                pmod0[i-lenWindow]) % mod0;
            h0 = (h0 + mod0) % mod0;
            h0 = (h0 + (data[i]+offsetVal)*pmod0[i]) %
                mod0;
            h1 = (h1 - (data[i-lenWindow]+offsetVal)*
                pmod1[i-lenWindow]);
            h1 = (h1 + (data[i]+offsetVal)*pmod1[i]);
            ans.push_back(iull((h0*pmod0[n-1-(i-
                lenWindow+1)])%mod0, h1*pmod1[n-1-(i-
                lenWindow+1)]));
        }
        return ans;
    };
};
```

6.2 KMP

```cpp
vi compute_lps(const string &pat){
    int m = pat.length();
    vi lps(m);
    int len = 0;
    for (int i = 1; i < m; i++){
        while(len > 0 && pat[i] != pat[len])
            len = lps[len-1];
        if (pat[i] == pat[len]) len++;
        lps[i] = len;
    }
    return lps;
}

// find all occurrences
vi kmp_search(const string &txt, const string &pat){
    int n = txt.length();
    int m = pat.length();
    if (m == 0) return {};
    vi lps = compute_lps(pat);
    vi occurrences;
    int j = 0;
    for (int i = 0; i < n; i++){
        while (j > 0 && txt[i] != pat[j])
            j = lps[j-1];
        if (txt[i] == pat[j]) j++;

        if (j == m) {
            occurrences.push_back(i-m+1);
            j = lps[j-1];
        }
    }
    return occurrences;
}

// find all occurrences (simpler version)
vi kmp_search(const string &txt, const string &pat){
    int n = txt.length(), m = pat.length();
    vi lps = compute_lps(pat + '#' + txt);
    vi occurrences;
    for (int i = 0; i < n+m+1; i++){
        if (lps[i] == pat.length())
            occurrences.push_back(i-m*2);
    }
    return occurrences;
}

// borda sao os prefixos que tambem sao sufixos
vi find_borders(const string &s){
    vi lps = compute_lps(s);
    int i = s.length()-1;

    vi ans;
    while (lps[i] > 0){
        ans.push_back(lps[i]);
        i = lps[i]-1;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

6.3 Suffix Array

Time: $\mathcal{O}(n \log n)$ Space: $\mathcal{O}(n)$

```cpp
struct SuffixArray {
  int sz;
  vi suff_ind, lcp;
  viii suffs;

  void radix_sort() {
    if (sz <= 1) return;
    viii suffs_new(sz);
    vi cnt(sz + 1, 0); /*rever esse tamanho*/

    for (auto& item : suffs) cnt[item.first.second]++;
    for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
    for (int i = sz - 1; i >= 0; --i) suffs_new[--cnt[suffs[i].first.second]] = suffs[i];

    cnt.assign(sz + 1, 0);
    for (auto& item : suffs_new) cnt[item.first.first]++;
    for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
    for (int i = sz - 1; i >= 0; --i) suffs[--cnt[suffs_new[i].first.first]] = suffs_new[i];
  }

  void build_lcp(vi& a) {
    lcp.assign(sz, 0);
    vi rank(sz);
    for (int i = 0; i < sz; ++i) rank[suff_ind[i]] = i;

    int h = 0;
    for (int i = 0; i < sz; ++i) {
      if (rank[i] == sz - 1) { h = 0; continue; }
      if (h > 0) h--;
      int j = suff_ind[rank[i] + 1];
      while (i + h < sz && j + h < sz && a[i + h] == a[j + h]) h++;
      lcp[rank[i] + 1] = h;
    }
  }

  void build(vi& a) {
    a.push_back(0);
    sz = a.size();
    suffs.resize(sz);
    suff_ind.resize(sz);
    vi equiv(sz);


    for (int i = 0; i < sz; ++i) suffs[i] = iii(ii(a[i],a[i]), i);
    radix_sort();
    for (int i = 1; i < sz; ++i) {
      auto [c,ci] = suffs[i];
      auto [p,pi] = suffs[i-1];
      equiv[ci] = equiv[pi] + (c > p);
    }

    for (int suflen = 1; suflen < sz; suflen *= 2) {
      for (int i = 0; i < sz; ++i) {
        suffs[i] = {{equiv[i], equiv[(i + suflen) % sz]}, i};
      }
      radix_sort();
      for (int i = 1; i < sz; ++i) {
        auto [c,ci] = suffs[i];
        auto [p,pi] = suffs[i-1];
        equiv[ci] = equiv[pi] + (c > p);
      }
    }

    for(int i = 0; i < sz; ++i) suff_ind[i] = suffs[i].second;
    build_lcp(a);

    a.pop_back();
    sz--;
    suff_ind.erase(suff_ind.begin());
    lcp.erase(lcp.begin());
  }
};
```

6.4 Suffix Automaton

```cpp
struct SAM {
    struct State {
        int len,link;
        ll cnt = 0;
        int first_occ=-1;
        map<char,int> next;
    };

    vector<State> st;
    int last;

    SAM(string s){
        st.push_back({0,-1,0,-1});
        last = 0;
        for (int i = 0; i < s.length(); i++){
            extend(s[i],i);
        }
        calc_cnt();
    }

    void extend(char c, int id){
        int cur = st.size();
        st.push_back({st[last].len+1,0,1,id});
        int p = last;
        while(p!=-1 && st[p].next.count(c)==0){
            st[p].next[c] = cur;
            p = st[p].link;
        }
        if (p == -1){
            st[cur].link = 0;
            last = cur;
            return;
        }

        int q = st[p].next[c];
        if (st[p].len+1 == st[q].len) {
            st[cur].link = q;
            last = cur;
            return;
        }
        int clone = st.size();
        st.push_back({
            st[p].len+1,
            st[q].link,
            0,
            st[q].first_occ,
            st[q].next
        });
        while (p!=-1 && st[p].next[c] == q){
            st[p].next[c] = clone;
            p = st[p].link;
        }
        st[q].link = st[cur].link = clone;
        last = cur;
    }

    void calc_cnt(){
        vi nodes(st.size());
        iota(nodes.begin(),nodes.end(),0);
        sort(nodes.begin(),nodes.end(),[&](int a, int b){
            return st[a].len > st[b].len;
        });

        for (int u : nodes){
            if (st[u].link != -1){
                st[st[u].link].cnt += st[u].cnt;
            }
        }
    }

    int count_occurrences(string t){
        int cur = 0;
        for (char c : t){
            if (st[cur].next.count(c) == 0) return 0;
            cur = st[cur].next[c];
        }
        return st[cur].cnt;
    }

    int first_occurrence(string t){
        int cur = 0;
        for (char c : t){
            if (!st[cur].next.count(c)) return -2;
            cur = st[cur].next[c];
        }
        return st[cur].first_occ-t.length()+1;
    }

    int distinct_substrings(){
        int ans = 0;
        for (int i = 1; i < st.size(); i++)
            ans += st[i].len - st[st[i].link].len;
        return ans;
    }

    vi distinct_substrings_perlen(int n){
        vi diff(n+2);
        for (int i = 1; i < st.size(); i++){
            int l = st[st[i].link].len+1;
            int r = st[i].len;
            diff[l]++; diff[r+1]--;
        }
        vi ans(n+1);
        ans[0] = diff[0];
        for (int i = 1; i <= n; i++){
            ans[i] = ans[i-1]+diff[i];
        }

        return ans;
    }

    vi dp;
    void calc_paths(int u){
        if (dp[u] != -1) return;
        dp[u]=1;
        for (auto [c,v] : st[u].next){
            calc_paths(v);
            dp[u] += dp[v];
        }
    }

    string find_kth(int k){
        dp.assign(st.size(),-1);
        calc_paths(0);
        int u = 0;
```

```cpp
        string ans = "";
        while(k>0){
            for (auto [c,v] : st[u].next){
                bool ok = false;
                if (k <= dp[v]){
                    ans += c;
                    u = v;
                    k--;
                    ok = true;
                    break;
                }
                if (!ok) k-=dp[v];
            }
        }
        return ans;
    }

        void calc_paths_with_repetitions(int u){
        if (dp[u] != -1) return;
        dp[u]=st[u].cnt;
        for (auto [c,v] : st[u].next){
            calc_paths_with_repetitions(v);
            dp[u] += dp[v];
        }
    }

    string find_kth_with_repetitions(int k){
        dp.assign(st.size(),-1);
        calc_paths_with_repetitions(0);
        int u = 0;
        string ans = "";
        while(k>0){
            for (auto [c,v] : st[u].next){
                bool ok = false;
                if (k <= dp[v]){
                    ans += c;
                    k-=st[v].cnt;
                    u = v;
                    ok = true;
                    break;
                }
                if (!ok) k-=dp[v];
            }
        }
        return ans;
    }
};
```

6.5 Z

$$z[i] := max(k)|s[0..k-1] = s[i..i+k-1]$$

Time: $\mathcal{O}(n+m)$ Space: $\mathcal{O}(n+m)$

```cpp
vi compute_z(const string &s) {
    int n = s.length();
    vi z(n);
    int l = 0, r = 0;

    for (int i = 1; i < n; i++) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i-1]);

        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;
}

vi find_occurrences(const string &txt, const string &
    pat){
    vi occurences;
    vi z = compute_z(pat + '#' + txt);
    int n = txt.length(), m = pat.length();
    for (int i = 0; i < n+m+1; i++){
        if (z[i] == m) occurences.push_back(i-m-1);
    }
    return occurences;
}
```

7 Math

7.1 Combinatorics (Pascal's Triangle)

Computes "n choose k". Requires factorials to be pre-computed. Time: $\mathcal{O}(\log ZAP)$

7.1.1 Combinatorial Analysis

Fundamental Counting Principles

- **Permutations:** The number of ways to arrange k items from a set of n distinct items.

$$P(n,k) = \frac{n!}{(n-k)!}$$

- **Combinations (Binomial Coefficient):** The number of ways to choose k items from a set of n distinct items, regardless of order.

$$\binom{n}{k} = C(n,k) = \frac{n!}{k!(n-k)!}$$

- **Combinations with Repetition (Stars and Bars):** The number of ways to choose k items of n types, allowing repetitions. Equivalently, the number of ways to distribute k identical balls into n distinct urns.

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

Binomial Coefficient Properties and Pascal's Triangle

- **Pascal's Triangle**

$$[n=0: \quad \binom{0}{0} \quad n=1: \quad \binom{1}{0} \quad \binom{1}{1} \quad n=2: \quad \binom{2}{0} \quad \binom{2}{1}$$

- **Stifel's Relation:** Each element in Pascal's Triangle is the sum of the two elements immediately above it.

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

- **Symmetry:** Elements of a row are symmetric with respect to the center. Choosing k elements is the same as choosing the $n-k$ elements to be left behind.

$$\binom{n}{k} = \binom{n}{n-k}$$

- **Row Sum:** The sum of all elements in row n of Pascal's Triangle (where the first row is $n=0$) is equal to 2^n.

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- **Hockey Stick Identity:** The sum of elements in a diagonal, starting at

$$\binom{r}{r}$$

and ending at

$$\binom{n}{r}$$

, is equal to the element in the next row and next column,

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

.

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

- **Binomial Theorem:**

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

- **Vandermonde's Identity:**

$$\sum_{j=0}^{k} \binom{m}{j}\binom{n}{k-j} = \binom{m+n}{k}$$

The easiest way to understand the identity is through a counting problem. Imagine you have a committee with m men and n women. How many ways can you form a subcommittee of k people?

Way 1 Direct Counting

You have a total of m+n people and need to choose k of them. The number of ways to do this is simply:

$$\binom{m+n}{k}$$

Way 2 Counting by Cases

We can divide the problem into cases, based on how many men (j) are chosen for the subcommittee.

Case 0: Choose 0 men and k women. The number of ways is

$$\binom{m}{0}\binom{n}{k}$$

.

Case 1: Choose 1 man and k-1 women. The number of ways is

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

.

Case j: Choose j men and $k-j$ women. The number of ways is

$$\binom{m}{j}\binom{n}{k-j}$$

.

Other Important Concepts

- **Catalan Numbers:** A sequence of natural numbers that occurs in various counting problems (e.g., number of binary trees, balanced parenthesis expressions).

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}$$

A commonly used combinatorial proof for the Catalan numbers involves counting the number of lattice (grid) paths from $(0,0)$ to (n,n) that do not cross above the diagonal $y=x$. Each such path consists of n rightward steps and n upward steps, and the Catalan number counts the number of these "Dyck paths" that never go above the diagonal.

- **Stirling Numbers of the Second Kind:** The number of ways to partition a set of n labeled objects into k non-empty unlabeled subsets. Denoted by $S(n,k)$ or

$$\{n\ k\}$$

.

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

The Stirling numbers of the second kind can also be computed recursively:

$$S(n,k) = k \cdot S(n-1,k) + S(n-1,k-1)$$

with the boundary conditions:

$$S(0,0)=1; \quad S(n,0)=0\,\text{for}\,n>0; \quad S(0,k)=0\,\text{for}\,k>0$$

- **Bell Number:** The Bell number B^n counts the total number of ways to partition a set of n labeled elements into any number (from 1 up to n) of non-empty, unlabeled subsets. It can also be written as a recurrence relation

$$B^n = \sum_{k=0}^{n} S(n,k)$$

- **Pigeonhole Principle:** If n items are put into m boxes, with $n > m$, then at least one box must contain more than one item.

```
// n escolhe k
// linha n, coluna k no triangulo (indexadas em 0)
int pascal(int n, int k){
    int num = fat[n];
    int den = (fat[k]*fat[n-k])%ZAP;
    return (num*expbin(den, ZAP-2))%ZAP;
}
```

7.2 Convolutions

7.2.1 AND convolution

$$c[k] = \sum_{i\&j=k} a[i]\cdot b[j]$$

```
vector<mint<MOD>> and_conv(vector<mint<MOD>> a, vector<
 mint<MOD>> b){
    int n = a.size(); // must be pow of 2
    for (int j = 1; j < n; j <<= 1) {
        for (int i = 0; i < n; i++) {
            if (i&j) {
                a[i^j] += a[i];
                b[i^j] += b[i];
            }
        }
    }

    for (int i = 0; i < n; i++) a[i] *= b[i];

    for (int j = 1; j < n; j <<= 1) {
        for (int i = 0; i < n; i++) {
            if (i&j) a[i^j] -= a[i];
        }
   }

    return a;
}
```

7.2.2 GCD convolution

$c[k] = \sum_{\gcd(i,j)=k} a[i]\cdot b[j]$

```
vector<mint<MOD>> gcd_conv(vi a, vi b){
    int n = (int)max(a.size(), b.size());
    a.resize(n);
    b.resize(n);
    vector<mint<MOD>> c(n);
    for (int i = 1; i < n; i++) {
        mint<MOD> x = 0;
        mint<MOD> y = 0;
        for (int j = i; j < n; j += i) {
            x += a[j];
            y += b[j];
        }
        c[i] = x*y;
    }
    for (int i = n-1; i >= 1; i--)
        for (int j = 2 * i; j < n; j += i)
            c[i] -= c[j];

    return c;
}
```

7.2.3 LCM convolution

$c[k] = \sum_{\mathrm{lcm}(i,j)=k} a[i]\cdot b[j]$

```
vector<mint<MOD>> lcm_conv(vi a, vi b){
    int n = (int)max(a.size(), b.size());
    a.resize(n);
```

```
    b.resize(n);
    vector<mint<MOD>> c(n), x(n), y(n);
    for (int i = 1; i < n; i++) {
        for (int j = i; j < n; j += i) {
            x[j] += a[i];
            y[j] += b[i];
        }
        c[i] = x[i]*y[i];
    }
    for (int i = 1; i < n; i++)
        for (int j = 2 * i; j < n; j += i)
            c[j] -= c[i];

    return c;
}
```

7.2.4 OR convolution

$$c[k] = \sum_{i|j=k} a[i] \cdot b[j]$$

```
vector<mint<MOD>> or_conv(vector<mint<MOD>> a, vector<
    mint<MOD>> b){
    int n = a.size(); // must be pow of 2
    for (int j = 1; j < n; j <<= 1) {
        for (int i = 0; i < n; i++) {
            if (i&j) {
                a[i] += a[i^j];
                b[i] += b[i^j];
            }
        }
    }

    for (int i = 0; i < n; i++) a[i] *= b[i];

    for (int j = 1; j < n; j <<= 1) {
        for (int i = 0; i < n; i++) {
            if (i&j) a[i] -= a[i^j];
        }
    }

    return a;
}
```

7.2.5 XOR convolution

$$c[k] = \sum_{i \oplus j=k} a[i] \cdot b[j]$$

```
void fwht(vector<mint<MOD>> &a, bool inv){
    int n = a.size(); // must be pow of 2
    for (int step = 1; step < n; step <<= 1){
        for (int i = 0; i < n; i += 2*step) {
            for (int j = i; j < i+step; j++){
                auto u = a[j];
                auto v = a[j+step];
                a[j] = u+v;
                a[j+step] = u-v;
            }
        }
    }
    if (inv) for (auto &x : a) x /= n;
}

vector<mint<MOD>> xor_conv(vector<mint<MOD>> a, vector<
    mint<MOD>> b){
    int n = a.size();
    fwht(a,0), fwht(b, 0);
    for (int i = 0; i < n; i++) a[i] *= b[i];
    fwht(a,1);
    return a;
}
```

7.3 Extended Euclid

Time: $\mathcal{O}(\log n)$.

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

7.4 Factorization

Time: $\mathcal{O}(\sqrt{n})$

```
// OBS: tem outras variantes mais rapidas no caderno da
    UDESC

// O(sqrt(n)) fatores repetidos
vi fatora(int n) {
    vi factors;
    for (int x = 2; x * x <= n; x++) {
        while (n % x == 0) {
            factors.push_back(x);
            n /= x;
        }
    }
    if (n > 1) factors.push_back(n);
    return factors;
}

// O(sqrt(n))
// Calcula a quantidade de divisores de um numero n.
int qtdDivisores(int n) {
    int ans = 1;
    for (int i = 2; i * i <= n; i += 2) {
        int exp = 0;
        while (n % i == 0) {
            n /= i; exp++;
        }
        if (exp > 0) ans *= (exp + 1);
        if (i == 2) i--;
    }
    if (n > 1) ans *= 2;
    return ans;
}

// O(sqrt(n))
// Calcula a soma de todos os divisores de um numero n.
ll somaDivisores(int n) {
    ll ans = 1;
    for (int i = 2; i * i <= n; i += 2) {
        if (n % i == 0) {
            int exp = 0;
            while (n % i == 0) {
                n /= i; exp++;
            }

            ll aux = expbin(i, exp + 1);
            ans *= ((aux - 1) / (i - 1));
        }
        if (i == 2) i--;
    }

    if (n > 1) ans *= (n + 1);
    return ans;
}
```

7.5 FFT - Fast Fourier Transform

Divide and conquer algorithm used for convolutions and polynomial multiplication. Vector size `a` is a power of 2. Time: $\mathcal{O}(n \log n)$ Space: $\mathcal{O}(n)$

```
void fft(vector<cd> &a, bool invert){
  int len = a.size();
  for(int i = 1, j = 0; i < len; i++){
    int bit = len >> 1;
    while(bit & j){
      j ^= bit;
      bit >>= 1;
    }
    j ^= bit;
    if(i < j) swap(a[i], a[j]);
  }
  for(int l = 2; l <= len; l <<= 1){
    double ang = 2*PI/l * (invert ? -1: 1);
    cd wd(cos(ang), sin(ang));
    for(int i = 0; i < len; i += l){
      cd w(1);
      for(int j = 0; j < l/2; j++){
        cd u = a[i+j], v = a[i+j+l/2];
        a[i+j] = u+w*v;
        a[i+j+l/2] = u-w*v;
        w *= wd;
      }
    }
  }
  if(invert){
    for(int i = 0; i < len; i++){
      a[i] /= len;
    }
  }
}
```

7.6 Inclusion-Exclusion Principle

TODO: rewrite math statement

```
// Exemplo:
// Contar numeros de 1 a n divisveis por uma lista de
    primos.
```

```cpp
int n;
vi primes;
int factors = primes.size();
int total_divisible = 0;

// Itera pelas bitmasks nao vazias de 'primes'
for (int i = 1; i < (1 << factors); i++) {
    int current_lcm = 1;
    int subset_size = 0;

    // calcula lcm do subconjunto
    for (int j = 0; j < factors; j++) {
        if (i & (1<<j)) {
            subset_size++;
            current_lcm = lcm(current_lcm, primes[j]);
            if (current_lcm > n) break;
        }
    }

    if (current_lcm > n) {
        continue;
    }

    int count = n / current_lcm;

    // Aplica o Principio da Inclusao-Exclusao:
    // Se o tamanho do subconjunto eh impar, adiciona.
    // Se o tamanho do subconjunto eh par, subtrai.
    if (subset_size & 1) {
        total_divisible += count;
    } else {
        total_divisible -= count;
    }
}
```

7.7 Legendre's formula

Computes the largest power of a prime p in n!

```cpp
int legendre(int n, int p){
  int ans = 0;
  while(n>0){
    n /= p; ans += n;
  }
  return ans;
}
```

7.8 Matrix template

A template for square matrices, used for solving linear recurrences with fast exponentiation, memory is on a 1D vector, but can be accessed with `A[i][j]` normally (custom `operator[]`)

```cpp
template<typename T>
struct mat{
  vector<T> m;
  int n;

  mat(int _n = 0, bool identity = false) : n(_n) {
    m.resize(n*n);
    if (!identity) return;
    for (int i = 0; i < n; i++)
      m[i*n+i] = 1;
  }

  mat& operator+=(const mat &o){
    for (int i = 0; i < n; i++){
      int ra = i*n;
      for (int j = 0; j < n; j++){
        m[ra+j] += o.m[ra+j];
      }
    }
    return *this;
  }
  mat& operator-=(const mat& o){
    for (int i = 0; i < n; i++){
      int ra = i*n;
      for (int j = 0; j < n; j++){
        m[ra+j] -= o.m[ra+j];
      }
    }
    return *this;
  }
  mat& operator*=(const mat& o){
    vector<T> ans(n*n);
    for (int i = 0; i < n; i++){
      for (int k = 0; k < n; k++){
        int ra = i*n, rb = k*n;
        for (int j = 0; j < n; j++){
          ans[ra+j] += m[ra+k] * o.m[rb+j];
        }
      }
    }
    this->m = ans;
    return *this;
  }
  friend mat operator+(mat a, const mat& b){
    return a+=b;
  }
  friend mat operator-(mat a, const mat& b){
    return a-=b;
  }
  friend mat operator*(mat a, const mat& b){
    return a*=b;
  }
  T* operator[](int i){
    return &m[i*n];
  }
  vector<T> operator*(const vector<T> &v){
    vector<T> ans(n);
    for (int i = 0; i < n; i++){
      int ra = i*n;
      for (int j = 0; j < n; j++)
        ans[i] += m[ra+j]*v[j];
    }
    return ans;
  }
  mat operator^(int e){
    return exp(*this, e);
  }
  static mat exp(mat b, int e){
    mat ans = mat(b.n,true);
    while(e>0){
      if(e&1) ans*=b;
      b*=b;
      e>>=1;
    }
    return ans;
  }
};
```

7.9 Mint

```cpp
template<ll MOD>
struct mint {
      ll val;
      mint(ll v = 0) {
            if (v < 0) v = v % MOD + MOD;
            if (v >= MOD) v %= MOD;
            val = v;
      }
      mint& operator+=(const mint& other) {
            val += other.val;
            if (val >= MOD) val -= MOD;
            return *this;
      }
      mint& operator-=(const mint& other) {
            val -= other.val;
            if (val < 0) val += MOD;
            return *this;
      }
      mint& operator*=(const mint& other) {
            val = (val * other.val) % MOD;
            return *this;
      }
      mint& operator/=(const mint& other) {
            val = (val * inv(other).val) % MOD;
            return *this;
      }
      friend mint operator+(mint a, const mint& b) {
            return a += b; }
      friend mint operator-(mint a, const mint& b) {
            return a -= b; }
      friend mint operator*(mint a, const mint& b) {
            return a *= b; }
      friend mint operator/(mint a, const mint& b) {
            return a /= b; }
      static mint power(mint b, ll e) {
            mint ans = 1;
            while (e > 0) {
                  if (e & 1) ans *= b;
                  b *= b;
                  e /= 2;
            }
            return ans;
      }
      static mint inv(mint n) { return power(n, MOD - 2);
            }
};
```

7.10 Modular Inverse

If m is prime, can use binary exponentiation to compute a^{p-2} (Fermat's Little Theorem).

This code works for non-prime m, as long as it is coprime to a.

Time: $\mathcal{O}(\log m)$

```cpp
int modInverse(int a, int m) {
      int x, y;
      int g = extendedGcd(a, m, x, y);
      if (g != 1) return -1;
      return (x % m + m) % m;
}
```

7.11 Number Theoretic Transform (NTT)

NTT is a fast algorithm (analogous to FFT) for polynomial multiplication modulo a special prime. It requires a prime modulus $p = c \cdot 2^k + 1$ (a "primitive root prime") and a primitive 2^k-th root of unity modulo p.

- **Prime Choices:** To use NTT, pick a modulus and a matching primitive root (see table below). For arbitrary moduli (e.g., $10^9 + 7$), multiply with several NTT-friendly primes and reconstruct with CRT (see `crt_multiply`).
- **Time Complexity:** $\mathcal{O}(n \log n)$ for polynomial multiplication.

7.11.1 NTT-Friendly Primes and Roots

NTT-friendly primes and their primitive roots:

- Mod: 998244353, Root: 3, Max N: 2^{23}
- Mod: 734003201, Root: 3, Max N: 2^{20}
- Mod: 167772161, Root: 3, Max N: 2^{25}
- Mod: 469762049, Root: 3, Max N: 2^{26}

Use the modulus as `MOD` and the root as `ROOT` when instantiating the NTT.

- For large/concrete moduli, see `crt_multiply` in the code for a multi-modulus solution with Chinese Remainder Theorem (CRT).

```cpp
    template<typename T, ll MOD, ll ROOT>
    void transform(vector<T>& a, bool invert) {
        int n = a.size();

        for (int i = 1, j = 0; i < n; i++) {
            int bit = n >> 1;
            for (; j & bit; bit >>= 1)
                j ^= bit;
            j ^= bit;
            if (i < j) swap(a[i], a[j]);
        }

        for (int len = 2; len <= n; len <<= 1) {
            T wlen = T::power(ROOT, (MOD - 1) / len);
            if (invert) wlen = T::inv(wlen);
            for (int i = 0; i < n; i += len) {
                T w = 1;
                for (int j = 0; j < len / 2; j++) {
                    T u = a[i + j], v = a[i + j + len / 2] * w;
                    a[i + j] = u + v;
                    a[i + j + len / 2] = u - v;
                    w *= wlen;
                }
            }
        }

        if (invert) {
            T n_inv = T::inv(n);
            for (T& x : a)
                x *= n_inv;
        }
    }

    template<typename T, ll MOD, ll ROOT>
    vector<ll> multiply(const vector<ll>& a, const vector<ll>& b) {
        vector<T> fa(a.begin(), a.end()), fb(b.begin(), b.end());
        int n = 1;
        while (n < a.size() + b.size()) n <<= 1;
        fa.resize(n);
        fb.resize(n);

        transform<T, MOD, ROOT>(fa, false);
        transform<T, MOD, ROOT>(fb, false);

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

        transform<T, MOD, ROOT>(fa, true);

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

    vector<ll> crt_multiply(const vector<ll>& a, const vector<ll>& b) {
        const ll mod1 = 998244353;
        const ll root1 = 3;
        using mint1 = mint<mod1>;
        vector<ll> ans1 = NTT::multiply<mint1, mod1, root1>(a, b);

        const ll mod2 = 1004535809;
        const ll root2 = 3;
        using mint2 = mint<mod2>;
        vector<ll> ans2 = NTT::multiply<mint2, mod2, root2>(a, b);

        int ans_size = a.size() + b.size() - 2;
        ll M1_inv_M2 = mint<mod2>::inv(mod1).val;

        vector<ll> final_result(ans_size + 1);
        for (int i = 0; i <= ans_size; ++i) {
            ll v1 = ans1[i];
            ll v2 = ans2[i];
            ll k = ((v2 - v1 + mod2) % mod2 * M1_inv_M2) % mod2;
            final_result[i] = v1 + k * mod1;
        }
        return final_result;
    }
}
```

7.12 Euler's Totient

Returns the amount of numbers smaller than n that are coprime to n. Time: $\mathcal{O}(\sqrt{n})$

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

8 Geometry

8.1 Convex hull - Graham Scan

Time: $\mathcal{O}(n \log n)$

```cpp
#define CLOCKWISE -1
#define COUNTERCLOCKWISE 1
#define INCLUDE_COLLINEAR 0 // pode mudar

struct Point {
    ll x, y;
    bool operator==(Point const& t) const {
        return x == t.x && y == t.y;
    }
};

struct Vec {
    int x, y, z;
};

Vec cross(Vec v1, Vec v2){
    int x = v1.y*v2.z - v1.z*v2.y;
    int y = -v1.x*v2.z + v1.z*v2.x;
    int z = v1.x*v2.y - v1.y*v2.x;
    return {x,y,z};
}

ll dist2(Point p1, Point p2){
    int dx = p1.x-p2.x;
    int dy = p1.y-p2.y;
    return dx*dx+dy*dy;
}

ll orientation(Point pivot, Point a, Point b){
    Vec va = {a.x-pivot.x, a.y-pivot.y, 0};
    Vec vb = {b.x-pivot.x, b.y-pivot.y, 0};
    Vec v = cross(va,vb);
    if (v.z < 0) return CLOCKWISE;
    if (v.z > 0) return COUNTERCLOCKWISE;
    return 0;
}

bool clock_wise(Point pivot, Point a, Point b) {
    int o = orientation(pivot, a, b);
    return o < 0 || (INCLUDE_COLLINEAR && o == 0);
}

bool collinear(Point a, Point b, Point c) { return orientation(a, b, c) == 0; }

vector<Point> convex_hull(vector<Point> &points, bool counterClockwise) {
    int n = points.size();
    Point pivot = *min_element(points.begin(), points.end(), [](Point a, Point b) {
        return ii(a.y, a.x) < ii(b.y, b.x);
    });
```

```cpp

    sort(points.begin(), points.end(), [&](Point a, Point b) {
        int o = orientation(pivot, a, b);
        if (o == 0) return dist2(pivot, a) < dist2(pivot, b);
        return o == CLOCKWISE;
    });

    if (INCLUDE_COLLINEAR) {
        int i = n-1;
        while (i >= 0 && collinear(pivot, points[i], points.back())) i--;
        reverse(points.begin()+i+1, points.end());
    }

    vector<Point> hull;
    for (auto p : points) {
        while (hull.size() > 1 && !clock_wise(hull[hull.size()-2], hull.back(), p))
        hull.pop_back();
        hull.push_back(p);
    }
    if (!INCLUDE_COLLINEAR && hull.size() == 2 && hull[0] == hull[1])
        hull.pop_back();

    if (counterClockwise && hull.size() > 1) {
        vector<Point> reversed_hull = hull;
        reverse(reversed_hull.begin() + 1, reversed_hull.end());
        return reversed_hull;
    }
    return hull;
}
```

8.2 Basic elements - geometry lib

- Basic elements for using the geometry lib, contains points, vector operations and distances between points, distance between point and segment, distance between segments, segment intersection check, orientation check (ccw).
- Always use long double for floating point. Only use floating point if indispensable.
- For a == b, use |a - b| < eps.!!!!

Time: $\mathcal{O}(1)$

8.2.1 Polygon Area

- **Heron's Formula for triangle area**:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

, where a, b, and c are the triangle sides and $s = (a+b+c)/2$

TODO Shoelace

- **Pick's Theorem for polygon area with integer coordinates**:

$$A = a + b/2 - 1$$

, where a is the number of integer coordinates inside the polygon and b is the number of integer coordinates on the polygon boundary. b can be calculated for each edge as

$$b = gcd(xi + 1 - xi, yi + 1 - yi) + 1$$

.

Polygon Area Time: $\mathcal{O}(n)$

8.2.2 Point in polygon

Sum of edge angles relative to the point must sum to 2pi Time: $\mathcal{O}(n \log n)$

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long double ld;
#define eps 1e-9
#define pi 3.141592653589
#define int long long int


struct pt {
    int x, y;
    int operator==(pt b) {
        return x == b.x && y == b.y;
    }
    int operator<(pt b) {
        if(x == b.x) return y < b.y;
        return x < b.x;
    }
    pt operator-(pt b) {
        return {x - b.x, y - b.y};
    }
    pt operator+(pt b) {
        return {x+b.x, y + b.y};
    }
};
int cross(pt u, pt v) {
    return u.x * v.y - u.y * v.x;
}
int dot(pt u, pt v) {
    return u.x * v.x + u.y * v.y;
}
ld norm(pt u) {
    return sqrt(dot(u, u));
}
ld dist(pt u, pt v) {
    return norm(u - v);
}
int ccw(pt u, pt v) { // cuidado com colineares!!!!
    return (cross(u, v) > eps)?1:((fabs(cross(u, v)) < eps)?0:-1);
}
int pointInSegment(pt a, pt u, pt v) { // checks if a lies in uv
    if(ccw(v - u, a - u)) return 0;
    vector<pt> pts = {a, u, v};
    sort(pts.begin(), pts.end());
    return pts[1] == a;
}
ld angle(pt u, pt v) { // angle between two vectors
    ld c = cross(u, v);
    ld d = dot(u,v);
    return atan2l(c, d);
}
int intersect(pt sa, pt sb, pt ra, pt rb) {// not sure if it works when one of the segments is a point
    pt s = sb - sa, r = rb - ra;
    if(pointInSegment(sa, ra, rb) || pointInSegment(sb, ra, rb) ||pointInSegment(ra, sa, sb) || pointInSegment(rb, sa, sb)) return 1;
    return !(ccw(s, ra - sa) == ccw(s, rb - sa) || ccw(r, sa - ra) == ccw(r, sb - ra));
}

ld polygonArea(vector<pt>& p) {// not signed (for signed area remove the absolute value at the end)
    ld area = 0;
    int n = p.size() - 1; // p[n] = p[0]
    for(int i = 0; i < n; i++) {
        area += cross(p[i], p[i + 1]);
    }
    return fabs(area)/2;
}
int pointInPolygon(pt a, vector<pt>& p) {// returns 0 for point in BOUNDARY, 1 for point in polygon and -1 for outside
    ld total = 0;
    int n = p.size() - 1;
    for(int i = 0; i < n; i++) {
        pt u = p[i] - a;
        pt v = p[i + 1] - a;
        if(fabs(dist(p[i], a) + dist(p[i + 1], a) - dist(p[i], p[i + 1])) < eps) {
            return 0;
        }
        total += angle(u, v);
    }

    return (fabs(fabs(total) - 2 * pi) < eps)?1:-1;
}



signed main() {
    int n, m; scanf("%lld %lld", &n, &m);
    vector<pt> p(n + 1);
    for(int i = 0; i < n; i++) {
        scanf("%lld %lld", &p[i].x, &p[i].y);
    }
    p[n] = p[0];

    while(m--) {
        pt a; scanf("%lld %lld", &a.x, &a.y);
        int ans = pointInPolygon(a, p);
        printf("%s\n", (ans > 0)?"INSIDE":(ans?"OUTSIDE":"BOUNDARY"));
    }
}
```