

Contents

1 Data Structures

1.1	Bit 2d
1.2	DSU - Disjoint Set Union
1.3	DSU - Binary Tree
1.4	Segment Tree
1.5	Sparse Table RMQ

2 Graphs

2.1	BFS 0-1
2.2	Dijkstra
2.3	Dinic - Flow/matchings
2.4	Floyd-Warshall
2.5	Hopcroft-Karp - Bipartite Matching
2.6	Hungarian
2.7	Kosaraju - SCCs
2.8	Kuhn - Bipartite Matching
2.9	Min cost flow
2.10	MST - Kruskal
2.11	MST - Prim

3 DP

3.1	Bin Packing
3.2	Broken Profile DP
3.3	Convex Hull Trick (CHT)
3.4	Edit Distance (Levenshtein)
3.5	Knapsack - 1D
3.6	Knapsack - 2D
3.7	LCS - Longest Common Subsequence
3.8	LiChao Tree
3.9	LIS - Longest Increasing Subsequence
3.10	SOSDP
3.11	Subset Sum

4 Trees

4.1	Sum of distances
4.2	Edge HLD
4.3	HLD - Heavy light decomposition
4.4	LCA - RMQ
4.5	LCA - binary lifting

5 Problemas clássicos

5.1	2SAT
5.2	Next Greater Element

6 Strings

6.1	Hashing
6.2	KMP
6.3	Suffix Array
6.4	Suffix Automaton
6.5	Z

7 Math

7.1	Combinatorics (Pascal's Triangle)
7.1.1	Combinatorial Analysis
7.2	Convolutions
7.2.1	AND convolution
7.2.2	GCD convolution
7.2.3	LCM convolution
7.2.4	OR convolution
7.2.5	XOR convolution
7.3	Extended Euclid
7.4	Factorization
7.5	FFT - Fast Fourier Transform
7.6	Inclusion-Exclusion Principle
7.7	Mint
7.8	Modular Inverse
7.9	Number Theoretic Transform (NTT)
7.9.1	NTT-Friendly Primes and Roots
7.10	Euler's Totient

8 Geometry

8.1	Convex hull - Graham Scan
8.2	Basic elements - geometry lib
8.2.1	Polygon Area
8.2.2	Point in polygon

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

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 #define MAX 1123
6
7 int bit[MAX][MAX], x, y;
```

```
8 void setbit(int i, int j, int delta) {
9     int j_;
10    while(i <= x) {
11        j_ = j;
12        while(j_ <= y) {
13            bit[i][j_] += delta;
14            j_ += j_ & -j_;
15        }
16        i += i & -i;
17    }
18 }
19 int getbit(int i, int j) {
20    int ans = 0;
21    int j_;
22    while(i) {
23        j_ = j;
24        while(j_) {
25            ans += bit[i][j_];
26            j_ -= j_ & -j_;
27        }
28        i -= i & -i;
29    }
30    return ans;
31 }
32
33 int main(void) {
34     int p;
35     while (scanf("%d %d %d", &x, &y, &p), x || y || p)
36     {
37         for(int i = 0 ; i <= x; i++)
38             for(int j = 0; j <= y; j++)
39                 bit[i][j] = 0;
40
41         int q;
42         scanf("%d", &q);
43         while(q--) {
44             char c;
45             scanf(" %c", &c);
46             int n, xi, yi, zi, wi;
47             if(c == 'A') {
48                 scanf(" %d %d %d", &n, &xi, &yi);
49                 xi++; yi++;
50                 setbit(xi, yi, n);
51             }
52             else {
53                 scanf(" %d %d %d %d", &xi, &yi, &zi, &
54                                 wi);
55                 xi++; yi++; zi++; wi++;
56                 if(xi > zi) swap(xi, zi);
57                 if(yi > wi) swap(yi, wi);
58                 ll ans = getbit(zi, wi) - getbit(zi, yi
59                                     - 1)
60                             - getbit(xi - 1, wi) + getbit(xi - 1,
61                                     yi - 1);
62             }
63             printf("%lld\n", ans * (ll) p);
64         }
65     }
66     return 0;
67 }
```

1.2 DSU - Disjoint Set Union

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

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

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

```

1 struct DSU {
2     vi p, sz;
3     DSU(int n) {
4         p.resize(n);
5         iota(p.begin(), p.end(), 0);
6         sz.assign(n, 1);
7     }
8     int find(int i) {
9         if (p[i] == i) return i;
10        return p[i] = find(p[i]);
11    }
12    bool unite(int u, int v) {
13        u = find(u);
14        v = find(v);
15        if (u == v) return false;
16        if (sz[u] < sz[v]) swap(u, v);
17        p[v] = u;
18        sz[u] += sz[v];
19        return true;
20    }
21 };

```

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: $\mathcal{O}(n)$

```

1 vi d;
2 vi_ii e;
3 vi ans;
4
5 int merged;
6 vi _p, _leaf, _wei;
7 vvi adj;
8 int _find(int u) { return _p[u] == u ? u : _p[u] = _find(_p[u]); }
9 void _union(int u, int v, int w){
10    u = _find(u);
11    v = _find(v);
12    int merge_ind = merged+n;
13    _p[u] = merge_ind;
14    _p[v] = merge_ind;
15    _leaf[merge_ind] = _leaf[u] + _leaf[v];
16    _wei[merge_ind] = max(_wei[u], _wei[v]);
17    adj[u].push_back(merge_ind);
18    adj[merge_ind].push_back(u);
19    adj[v].push_back(merge_ind);
20    adj[merge_ind].push_back(v);
21    merged++;
22 }
23 void make(){
24     _p = vi(2*n);
25     for(int i = 0; i < 2*n; i++) _p[i] = i;
26     _leaf = vi(2*n, 1);
27     _wei = vi(2*n);
28     for(int i = 0; i < n; i++) _wei[i] = d[i];
29     merged = 0;
30     adj = vvi(2*n);
31 }
32 void dfs(int u, int p){
33     for(auto &v: adj[u]){
34         if(v == p) continue;

```

```

36         ans[v] = ans[u] + (_leaf[u] - _leaf[v])*_wei[u];
37     }
38 }
39 }

47 }
48
49 node merge(node a, node b){
50     return {
51         max(max(a.seg, b.seg), a.suf + b.pre),
52         max(a.pre, a.sum + b.pre),
53         max(b.suf, b.sum + a.suf),
54         a.sum+b.sum
55     };
56 }
57
58 void init (vi &a){
59     int n = a.size();
60     size = 1;
61     while (size < n) size *= 2;
62     nodes.assign(2*size-1, NEUTRAL);
63     hasLazy.assign(2*size-1, false);
64     lazy.assign(2*size-1, 0);
65     build(0,0,size,a);
66 }
67
68 void build(int x, int lx, int rx, vi &a){
69     if (rx-lx == 1){
70         if (lx < a.size()) nodes[x] = single(a[lx]);
71         return;
72     }
73     int mx = (lx+rx)/2;
74     build(2*x+1,lx,mx,a);
75     build(2*x+2,mx,rx,a);
76     nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
77 }
78
79 void set(int i, int v, int x, int lx, int rx){
80     if (rx-lx == 1){
81         nodes[x] = single(v);
82         return;
83     }
84     int mx = (lx+rx)/2;
85     if (i < mx) set(i, v, 2*x+1, lx, mx);
86     else set(i, v, 2*x+2, mx, rx);
87     nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
88 }
89
90 void set(int i, int v){
91     set(i, v, 0, 0, size);
92 }
93
94 void rangeUpdate(int l, int r, int v){
95     rangeUpdate(l,r,v,0,0,size);
96 }
97
98 void rangeUpdate(int l, int r, int v, int x, int lx,
99                  int rx){
100    unlazy(x,lx);
101    if (rx-lx < 1 || rx <= l || lx >= r) return;
102    if (l <= lx && rx <= r) return propagate(x,lx,rx,v);
103    int mx = (lx+rx)/2;
104    rangeUpdate(l,r,v,2*x+1,lx,mx);
105    rangeUpdate(l,r,v,2*x+2,mx,rx);
106    nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
107 }
108
109 node query(int l, int r){
110     return query(l,r,0,0,size);
111 }
112
113 node query(int l, int r, int x, int lx, int rx){
114     unlazy(x,lx,rx);

```

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

```

1 struct segtree {
2     int size;
3     vector<node> nodes;
4     vector<bool> hasLazy;
5     vector<int> lazy;
6
7     struct node {
8         int seg, pre, suf, sum;
9     };
10
11     node NEUTRAL = {0,0,0,0};
12
13     void debug(){
14         if (nodes.empty() || size == 0) {
15             cout << "[Empty Tree]\n"; return;
16         }
17
18         string indent = "...";
19         function<void(int, int, int, string)> print_dfs;
20
21         print_dfs = [&](int x, int lx, int rx, string
22                         prefix) {
23             cout << prefix << "[" << lx << ", " << rx << "]";
24
25             // debug node
26             node a = nodes[x];
27             cout << "{ ";
28             cout << "seg: " << a.seg << ', ';
29             cout << "pre: " << a.pre << ', ';
30             cout << "suf: " << a.suf << ', ';
31             cout << "sum: " << a.sum << ', ';
32             cout << "hasLazy: " << hasLazy[x] << ', ';
33             cout << "lazy: " << lazy[x] << ', ';
34             cout << "}";
35             cout << endl;
36
37             if (rx-lx <= 1) return;
38
39             int mx = (lx+rx)/2;
40             print_dfs(2*x+1, lx, mx, prefix + indent);
41             print_dfs(2*x+2, mx, rx, prefix + indent);
42         };
43         print_dfs(0, 0, size, "");
44
45         node single(int v){
46             return {v,v,v,v};

```

```

115     if (rx-lx < 1 || rx <= 1 || lx >= r) return NEUTRAL;
116     ;
117     if (l <= lx && rx <= r) return nodes[x];
118     int mx = (lx+rx)/2;
119     node left = query(l,r,2*x+1,lx,mx);
120     node right = query(l,r,2*x+2,mx,rx);
121     return merge(left,right);
122 }
123 void unlazy(int x, int lx, int rx){
124     if (hasLazy[x]){
125         propagate(x, lx, rx, lazy[x]);
126         hasLazy[x] = false;
127     }
128 }
129 void propagate(int x, int lx, int rx, int v){
130     nodes[x].sum = (rx-lx)*v;
131     nodes[x].seg = max((rx-lx)*v,0ll);
132     nodes[x].pre = max((rx-lx)*v,0ll);
133     nodes[x].suf = max((rx-lx)*v,0ll);
134     if (rx-lx > 1){
135         lazy[2*x+1] = v;
136         lazy[2*x+2] = v;
137         hasLazy[2*x+1] = true;
138         hasLazy[2*x+2] = true;
139     }
140 }
141 }
142 };

```

1.5 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.

```

1 struct SparseTable {
2     vector<vector<i>> st;
3
4     void build(const vi &a) {
5         int n = a.size();
6         int max_log = __bit_width(n);
7         st.assign(max_log, vector<i>(n));
8         for (int i = 0; i < n; i++) {
9             st[0][i] = {a[i], i};
10        }
11        for (int i = 1; i < max_log; i++) {
12            for (int j = 0; j + (1 << i) <= n; j++) {
13                // Combine the two halves
14                st[i][j] = std::min(st[i-1][j], st[i-1][j + (1
15                    << (i-1))]);
16            }
17        }
18    }
19
20    // Returns min value and index in range [l, r]
21    // inclusive
22    ii min(int l, int r) {
23        int len = r - l + 1;
24        int k = __bit_width(len) - 1;
25        return std::min(st[k][l], st[k][r - (1<<k) + 1]);
26    };

```

2 Graphs

2.1 BFS 0-1

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

```

1 vi bfs01(int s){
2     vi d(n, INF);
3     d[s] = 0;
4     deque<int> q;
5     q.push_front(s);
6     while(!q.empty()){
7         int u = q.front(); q.pop_front();
8         for (auto [v,w] : adj[u]){
9             if (d[u]+w < d[v]){
10                 d[v] = d[u] + w;
11                 if (w == 1) q.push_back(v);
12                 else q.push_front(v);
13             }
14         }
15     }
16     return d;
17 }

```

2.2 Dijkstra

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

```

1 void dijkstra(int s){
2     int d, u, v;
3     dist = vi(n, INF);
4     dist[s] = 0;
5     priority_queue<ii, vii, greater<ii>> pq;
6     pq.emplace(0,s);
7     while(!pq.empty()){
8         auto [d,u] = pq.top(); pq.pop();
9         if (d > dist[u]) continue;
10        for (auto [v,w] : adj[u]){
11            if (dist[v] > dist[u] + w){
12                dist[v] = dist[u] + w;
13                pq.emplace(dist[v], v);
14            }
15        }
16    }
17 }

```

2.3 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 $\mathcal{O}(E\sqrt{V})$.

```

1 struct Dinic {
2     struct Edge {
3         int u, v;
4         ll cap, flow = 0;

```

```

5         Edge(int u, int v, ll cap) : u(u), v(v), cap(cap)
6         {}
7     };
8     const ll flow_inf = 1e18;
9     vector<Edge> edges;
10    vvi adj;
11    int n, m = 0;
12    int s, t;
13    vi level, ptr;
14    queue<int> q;
15
16    Dinic(int n): n(n) {
17        adj.resize(n);
18        level.resize(n);
19        ptr.resize(n);
20    }
21
22    void add_edge(int u, int v, ll cap) {
23        edges.emplace_back(u,v,cap);
24        edges.emplace_back(v,u,0);
25        adj[u].push_back(m++);
26        adj[v].push_back(m++);
27    }
28
29    bool bfs(ll delta){
30        queue<int> q;
31        q.push(s);
32        while(!q.empty()){
33            int u = q.front(); q.pop();
34            for (int id : adj[u]){
35                auto &e = edges[id];
36                if (e.cap - e.flow < delta) continue;
37                if (level[e.v] != -1) continue;
38                level[e.v] = level[u]+1;
39                q.push(e.v);
40            }
41        }
42        return level[t] != -1;
43    }
44
45    ll dfs(int u, ll pushed) {
46        if (pushed == 0) return 0;
47        if (u == t) return pushed;
48        for (int &cid = ptr[u]; cid < (int)adj[u].size();
49             cid++){
50            int id = adj[u][cid];
51            auto &e = edges[id];
52            if (level[u]+1 != level[e.v]) continue;
53            ll tr = dfs(e.v,min(pushed, e.cap - e.flow));
54            if (tr == 0) continue;
55            e.flow += tr;
56            edges[id^1].flow -= tr;
57            return tr;
58        }
59        return 0;
60    }
61
62    ll maxflow(int s, int t){
63        this->s = s; this->t = t;
64        ll max_c = 0;
65        for (auto &e : edges) max_c = max(max_c, e.cap);
66
67        ll delta = 1;
68        while(delta <= max_c) delta <= 1;
69        delta >>= 1;
70
71        ll f = 0;
72        for (;delta > 0; delta >>= 1){
73            while(true){

```

```

73     fill(level.begin(), level.end(), -1);
74     level[s] = 0;
75     if (!bfs(delta)) break;
76     fill(ptr.begin(), ptr.end(), 0);
77     while(ll pushed = dfs(s, flow_inf)) f += pushed;
78   }
79 }
80 return f;
81 }
82
83 // call constructor with (n1+n2+2) beforehand (don't
84 // add edges manually)
85 // assumes pairs are 1-indexed
86 vii maxmatchings(int n1, int n2, const vvi& pairs){
87   for (int i = 1; i <= n1; i++)
88     add_edge(0, i, 1);
89
90   for (int i = 1; i <= n2; i++)
91     add_edge(i+n1, n1-1, 1);
92
93   for (auto &[u,v] : pairs)
94     add_edge(u, v+n1, 1);
95
96   maxflow(0, n-1);
97
98   vii matchings;
99   for (auto &e : edges){
100     if (e.u >= 1 && e.u <= n1 && e.flow == 1 && e.v >
101         n1){
102       matchings.emplace_back(e.u, e.v-n1);
103     }
104   }
105   return matchings;
106 }
107
108 vii mincut(int s, int t){
109   maxflow(s,t);
110   queue<int> q; q.push(s);
111   vector<bool> reachable(n);
112   reachable[s] = true;
113   while(!q.empty()){
114     int u = q.front(); q.pop();
115     for (auto &id : adj[u]){
116       int v = edges[id].v;
117       if (edges[id].cap - edges[id].flow > 0 && !
118           reachable[v]) {
119         reachable[v] = true;
120         q.push(v);
121       }
122     }
123   }
124   vii minCutEdges;
125   for (int i = 0; i < m; i += 2) {
126     const Edge& edge = edges[i];
127     if (reachable[edge.u] && !reachable[edge.v]) {
128       minCutEdges.emplace_back(edge.u, edge.v);
129     }
130   }
131   return minCutEdges;
132 }
133 };

```

2.4 Floyd-Warshall

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

```

1  vvi d(n, vi(n, INF));
2  void floyd_warshall(){
3    for (int k = 0; k < n; k++)
4      for (int i = 0; i < n; i++)
5        for (int j = 0; j < n; j++)
6          d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
7  }
8

```

2.5 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: $\mathcal{O}(E\sqrt{V})W$

```

1  int n, m, k;
2  vvi adj;
3  vi p, dist; /*p is in matching for [0, n[ and parent
               for [n, n+m[*/
4
5  int bfs(){
6    queue<int> q;
7    dist = vi(n+m, inf);
8    for(int i = 0; i < n; i++){
9      if(p[i] == -1) q.push(i), dist[i] = 0;
10 }
11 int min_dist_match = inf;
12 while(!q.empty()){
13   int u = q.front(); q.pop();
14   if(dist[u] > min_dist_match) continue;
15   for(auto v: adj[u]){
16     if(p[v] == -1) min_dist_match = dist[u];
17     else if(dist[p[v]] == inf){
18       dist[p[v]] = dist[u] + 1;
19       q.push(p[v]);
20     }
21   }
22 }
23 return min_dist_match != inf;
24 }
25
26 int dfs(int u){
27   for(auto v: adj[u]){
28     if(p[v] == -1 || (dist[u]+1 == dist[p[v]] && dfs(p[v]))){
29       p[v] = u;
30       p[u] = 1;
31       return true;
32     }
33   }
34   dist[u] = inf;
35   return false;
36 }
37
38 int hopkarp(){
39   p = vi(n+m, -1);
40   int matchings = 0;
41   while(bfs()){
42     for(int i = 0; i < n; i++){
43       if(p[i] == -1 && dfs(i)) matchings++;
44     }
45   }

```

```

46   return matchings;
47 }
48
49 void create(){
50   adj = vvi(n+m);
51   for(int i = 0; i < k; i++){
52     int u, v;
53     cin >> u >> v; u--; v--;
54     v += n;
55     adj[u].push_back(v);
56   }
57 }

```

2.6 Hungarian

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

```

1 // cost should be (cost[worker][job])
2 pair<int,vii> hungarian(int n, int m, const vvi &cost)
3 {
4   if (n == 0) return {0, {}};
5   int N = max(n, m);
6
7   vi u(N+1), v(N+1), p(N+1), way(N+1);
8
9   const int INF = 1e9;
10  for (int i = 1; i <= n; ++i) {
11    p[0] = i;
12    int j0 = 0;
13    vi minv(N + 1, INF);
14    vector<bool> used(N + 1, false);
15
16    do {
17      used[j0] = true;
18      int i0 = p[j0], delta = INF, j1;
19
20      for (int j = 1; j <= N; ++j) {
21        if (!used[j]) {
22          int cur = cost[i0-1][j-1] - u[i0] - v[j];
23          if (cur < minv[j]) {
24            minv[j] = cur;
25            way[j] = j0;
26          }
27        if (minv[j] < delta) {
28          delta = minv[j];
29          j1 = j;
30        }
31      }
32    }
33
34    for (int j = 0; j <= N; ++j) {
35      if (used[j]) {
36        u[p[j]] += delta;
37        v[j] -= delta;
38      } else {
39        minv[j] -= delta;
40      }
41    }
42    j0 = j1;
43  } while (p[j0] != 0);
44
45  do {
46    int j1 = way[j0];
47    p[j0] = p[j1];
48    j0 = j1;

```

```

49     } while (j0);
50 }
51
52 int total_cost = 0;
53 for (int j = 1; j <= m; ++j) {
54     if (p[j] != 0) {
55         total_cost += cost[p[j] - 1][j - 1];
56     }
57 }
58
59 // {worker, job}[] 0-indexed
60 vii matchings;
61 for (int j = 1; j <= m; ++j) {
62     if (p[j] != 0) {
63         matchings.push_back({p[j] - 1, j - 1});
64     }
65 }
66 return {total_cost, matchings};
67 }

```

2.7 Kosaraju - SCCs

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

```

1 void dfs1(int u){
2     vis[u] = 1;
3     for (auto v : adj[u]){
4         if (!vis[v]) dfs1(v);
5     }
6
7     ts.push_back(u);
8 }
9
10 void dfs2(int u, int c){
11     scc[u] = c;
12     for (auto v : adjT[u])
13         if (!scc[v]) dfs2(v,c);
14 }
15
16
17 // usage
18 for (int i = 0; i < n; i++)
19     if (!vis[i]) dfs1(i);
20
21 reverse(ts.begin(), ts.end());
22
23 int c = 1;
24 for (auto u : ts)
25     if (!scc[u]) dfs2(u,c++);

```

2.8 Kuhn - Bipartite Matching

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

```

1 int matchings;
2 vi p, vis;
3 vii match;
4
5 int dfs(int u){
6     if(vis[u]) return 0;
7     vis[u] = 1;
8     for(auto v: adj[u]){
9         if(p[v] == -1 || dfs(p[v])){
10             p[v] = u;
11             return 1;
12         }
13     }
14     return 0;
15 }
16
17 void kuhn(){
18     matchings = 0;
19     p = vi(n+m, -1);
20     for(int i = 0; i < n; i++){
21         vis = vi(n, 0);
22         matchings += dfs(i);
23     }
24     for(int i = n; i < n+m; i++){
25         if(p[i] != -1) match.push_back(ii(p[i], i));
26     }
27 }
28
29 void create(){
30     adj = vvi(n+m);
31     for(int i = 0; i < k; i++){
32         int u, v;
33         cin >> u >> v; u--; v--;
34         adj[u].push_back(v+n);
35     }
36 }

```

2.9 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 $\mathcal{O}(EV + FE \log V)$.

```

1 struct MinCostFlow {
2     struct Edge {
3         int to, capacity, rev;
4         ll cost;
5     };
6     int n;
7     vector<vector<Edge>> adj;
8
9     MinCostFlow(int _n) : n(_n), adj(_n) {}
10
11     void add_edge(int from, int to, int cap, ll cost){
12         adj[from].push_back({to,cap,(int)adj[to].size(),
13                             cost});
14         adj[to].push_back({from,0,(int)adj[from].size()-1,
15                           -cost});
16     }
17
18     // O(FE log(V))
19     lli min_cost_flow(int s, int t, int targetFlow) {
20         int flow = 0;
21         ll total_cost = 0;
22         vll dist, h(n);
23         vi pv, pe;
24
25         // needed only if negative costs exists
26         spfa(s, h, pv, pe);
27
28         while (flow < targetFlow) {
29             dijkstra(s, h, dist, pv, pe);
30         }
31     }
32 }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96

```

```

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;
}

```

```

97 priority_queue<lli, vector<lli>, greater<lli>> pq;
98 pq.emplace(0, s);
99 while (!pq.empty()) {
100     auto [d, u] = pq.top(); pq.pop();
101     if (d > dist[u]) continue;
102
103     for (int i = 0; i < adj[u].size(); i++) {
104         Edge &e = adj[u][i];
105         if (e.capacity <= 0) continue;
106         int v = e.to;
107
108         ll reduced_cost = e.cost + h[u] - h[v];
109         if (dist[u] != INF && dist[v] > dist[u] +
110             reduced_cost) {
111             dist[v] = dist[u] + reduced_cost;
112             pv[v] = u;
113             pe[v] = i;
114             pq.push({dist[v], v});
115         }
116     }
117 }
118 }
119 }
120 // usage
121 int nodes = 302; // amount of nodes in the network
122 MinCostFlow mcf(nodes);
123
124 for (int i = 0; i < 150; i++){
125     mcf.add_edge(0, i+1, 1, 0); // source to node
126     mcf.add_edge(i+151, nodes-1, 1, 0); // node to sink
127 }
128
129 for (int i = 0; i < n; i++){
130     int a, b, c; cin >> a >> b >> c;
131     mcf.add_edge(a, b+150, 1, -c); // edges in between (-
132     c to maximize the cost)
133 }
134
135 // final max cost is -cost
136 auto [cost, flow] = mcf.min_cost_flow(0,nodes-1,150);

```

2.10 MST - Kruskal

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

```

1 vector<pai<int,ii>> edges; // [weight, (u,v)]
2 int kruskal(int n){
3     int cost = 0;
4     DSU dsu(n); // n is the num of vertices
5     sort(edges.begin(), edges.end());
6     for (auto &[w,uv] : edges){
7         auto [u,v] = uv;
8         if (dsu.unite(u,v)) cost += w;
9     }
10    return cost;
11 }

```

2.11 MST - Prim

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

```

1 vvii adj, mst;
2 vi taken;

```

```

3
4 int prim(){
5     priority_queue<iii, vector<iii>, greater<iii>> pq;
6     taken[0] = 1;
7     for (auto [w,v] : adj[0]){
8         if (!taken[v]) pq.push({w, {0,v}});
9     }
10
11     int cost = 0;
12     while (!pq.empty()){
13         auto [w,u] = pq.top(); pq.pop();
14         auto [p,u] = pu;
15         if (!taken[u]){
16             cost += w;
17             mst[p].emplace_back(w,u);
18             mst[u].emplace_back(w,p);
19             taken[u] = 1;
20             for (auto [w,v] : adj[u]){
21                 if (!taken[v]){
22                     pq.push({w,{u,v}});
23                 }
24             }
25         }
26     }
27     return cost;
28 }

```

3 DP

3.1 Bin Packing

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

```

1 vi w(n);
2
3 vector<ii> dp(1<<n, ii(INF,0));
4 // dp[i] = for the subset i(bitmap) (A,B) is the pair
5 // where
5 // A - the min number of knapsacks to store this subset
6 // B - the min size of a used knapsack
7
8 dp[0] = ii(0,INF);
9 for (int subset = 1; subset < (1<<n); subset++){
10    for (int item = 0; item < n; item++){
11        if (!(subset >> item) & 1) continue;
12        int prevsubset = subset - (1<<item);
13        ii prev = dp[prevsubset];
14
15        if (prev.second + w[item] <= x) {
16            // can fill the knapsack, fill it
17            dp[subset] = min(dp[subset], ii(prev.first, prev.
18                                second+w[item]));
19        } else {
20            // cant fill the knapsack, create a new one
21            dp[subset] = min(dp[subset], ii(prev.first+1, w[
22                                item]));
23        }
24    cout << dp[(1<<n)-1].first << endl;
25 }

```

3.2 Broken Profile DP

Solves the problem of counting how many ways to fill an $n \times m$ grid using 1×2 tiles. This technique can be used whenever the state dependence is only on the previous state (column). Time: $\mathcal{O}(mn2^n)$ Space: $\mathcal{O}(mn2^n)$

```

1 int dp[1002][12][1024];
2 dp[0][0][0] = 1;
3
4 for (int i = 0; i < m; i++){
5     for (int j = 0; j < n; j++){
6         for (int mask = 0; mask < (1<<n); mask++){
7             if (mask & (1<<j)){
8                 int nxt_mask = mask - (1<<j);
9                 dp[i][j+1][nxt_mask] += dp[i][j][mask];
10                dp[i][j+1][nxt_mask] %= M;
11            } else {
12                int q = mask + (1 << j);
13                dp[i][j+1][q] += dp[i][j][mask];
14                dp[i][j+1][q] %= M;
15                if (j < n-1 && (mask & (1<<(j+1))) == 0){
16                    q = mask + (1 << (j+1));
17                    dp[i][j+1][q] += dp[i][j][mask];
18                    dp[i][j+1][q] %= M;
19                }
20            }
21        }
22    }
23
24    for (int p = 0; p < (1<<n); p++){
25        dp[i+1][0][p] = dp[i][n][p];
26    }
27 }

```

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

```

1 struct CHT {
2     struct Line { //  $y = mx + c$ 
3         int m, c;
4         Line(int m, int c) : m(m), c(c) {}
5         int val(int x){
6             return m*x + c;
7         }
8         int floorDiv(int num, int den) {
9             if (den < 0) num = -num, den = -den;
10            if (num >= 0) return num / den;
11            else return - ( (-num + den - 1) / den );
12        }
13        int ceilDiv(int num, int den) {
14            if (den < 0) num = -num, den = -den;
15            if (num >= 0) return (num + den - 1) / den;
16            else return - ( (-num) / den );
17        }
18        int intersect(Line l){
19            //  $m_1x + c_1 = m_2x + c_2$ 
20            //  $x = (c_2 - c_1)/(m_1 - m_2)$ 
21            // if slopes are increasing, use floor div
22            return ceilDiv(l.c - c, m - l.m);
23        }
24    };
25
26    deque<pair<Line, int>> dq;
27
28    void insert(int m, int c){
29        Line newLine(m, c);
30        if (!dq.empty() && newLine.m == dq.back().first.m)
31        {
32            // If slopes increasing, change to <=
33            if (newLine.c >= dq.back().first.c) return;
34            else dq.pop_back();
35        }
36        // if slopes increasing, change to <=
37        while (dq.size() > 1 && dq.back().second >= dq.back()
38            .first.intersect(newLine)){
39            dq.pop_back();
40        }
41        if (dq.empty()){
42            // assuming queries are positive numbers, may
43            // change to -INF or +INF if needed
44            dq.emplace_back(newLine, 0);
45        }
46        dq.emplace_back(newLine, dq.back().first.intersect(
47            newLine));
48
49        // dont use query and queryNonMonotonicValues in the
50        // same problem
51        int query(int x){
52            else break;
53        }
54        return dq[0].first.val(x);
55    }
56
57
58    int queryNonMonotonicValues(int x){
59        int l=0, r=dq.size()-1, ans=0;
60        while (l <= r) {
61            int mid = (l+r)>>1;
62            if (dq[mid].second <= x) {
63                ans = mid;
64                l = mid + 1;
65            } else {
66                r = mid - 1;
67            }
68        }
69        return dq[ans].first.val(x);
70    }
71 }
72
73 void solve(){
74     int n, c; cin >> n >> c;
75     vi h(n);
76     for (auto &x : h) cin >> x;
77
78     vi dp(n);
79     dp[0] = 0;
80     CHT cht;
81     cht.insert(-2*h[0], h[0]*h[0]);
82     for (int i = 1; i < n; i++){
83         dp[i] = cht.query(h[i]) + c + h[i]*h[i];
84         cht.insert(-2*h[i], h[i]*h[i] + dp[i]);
85     }
86     cout << dp[n-1] << endl;
87 }
```

3.4 Edit Distance (Levenshtein)

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

```

1 vvi dp(n+1, vi(m+1));
2 for (int i = 0; i <= n; i++) dp[i][0] = i;
3 for (int i = 0; i <= m; i++) dp[0][i] = i;
4 for (int i = 1; i <= n; i++){
5     for (int j = 1; j <= m; j++){
6         dp[i][j] = min(
7             min(dp[i][j-1]+1, dp[i-1][j]+1),
8             dp[i-1][j-1]+(s[i-1] != t[j-1])
9         );
10    }
11 }
```

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: $\mathcal{O}(W)$

```

1 vi dp(W+1);
2 for (int i = 0; i < n; i++){
3     for (int j = W; j >= w[i]; j--){
4         dp[j] = max(dp[j], v[i] + dp[j-w[i]]);
5     }
6 }
```

```

3     for (int j = W; j >= w[i]; j--){
4         dp[j] = max(dp[j], v[i] + dp[j-w[i]]);
5     }
6 }
```

3.6 Knapsack - 2D

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

```

1 vvi dp(n+1, vi(W+1));
2 for (int c = 1; c <= W; c++){
3     for (int i = 1; i <= n; i++){
4         dp[i][c] = dp[i-1][w];
5         if (c-w[i-1] >= 0) {
6             dp[i][c] = max(dp[i][c], dp[i-1][c-w[i-1]] + v[i-1]);
7         }
8     }
9 }
```

3.7 LCS - Longest Common Subsequence

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

```

1 vvi dp(n+1, vi(m+1));
2 vvii p(n+1, vii(m+1));
3
4 for (int i = 1; i <= n; i++){
5     for (int j = 1; j <= m; j++){
6         if (a[i-1] == b[j-1]){
7             dp[i][j] = dp[i-1][j-1]+1;
8             p[i][j] = {i-1, j-1};
9         } else if (dp[i][j-1] > dp[i-1][j]){
10            dp[i][j] = dp[i][j-1];
11            p[i][j] = {i, j-1};
12        } else {
13            dp[i][j] = dp[i-1][j];
14            p[i][j] = {i-1, j};
15        }
16    }
17 }
18
19 ii pos = ii(n,m);
20 stack<int> st;
21 while (pos != ii(0,0)){
22     auto [i,j] = pos;
23     if (p[i][j] == ii(i-1, j-1)) st.push(a[i-1]);
24     pos = p[i][j];
25 }
26 cout << st.size() << endl;
27 while (!st.empty()){
28     cout << st.top() << ' ';
29     st.pop();
30 }
31 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.

```

1 // Li Chao tree for minimum (or maximum) over domain [L, R].
2 // T should support +, *, comparisons.
3 // For integer x use eps = 0 and discrete mid+1 splitting;
4 // For floating use eps > 0 and continuous splitting without +1.
5 template<typename T>
6 struct lichao_tree {
7     // if max lichao, change to ::min()
8     static const T identity = numeric_limits<T>::max();
9
10    struct Line {
11        T m, c;
12        Line() {
13            m = 0;
14            c = identity;
15        }
16        Line(T m, T c) : m(m), c(c) {}
17        T val(T x) { return m * x + c; }
18    };
19
20    struct Node {
21        Line line;
22        Node *lc, *rc;
23        Node() : lc(0), rc(0) {}
24    };
25
26    T L, R, eps;
27    deque<Node> buffer;
28    Node* root;
29
30    Node* new_node() {
31        buffer.emplace_back();
32        return &buffer.back();
33    }
34
35    lichao_tree() {}
36
37    lichao_tree(T _L, T _R, T _eps) {
38        init(_L, _R, _eps);
39    }
40
41    void clear() {
42        buffer.clear();
43        root = nullptr;
44    }
45
46    void init(T _L, T _R, T _eps) {
47        clear();
48        L = _L;
49        R = _R;
50        eps = _eps;
51        root = new_node();
52    }
53
54    void insert(Node* &cur, T l, T r, Line line) {
55        if (!cur) {
56            cur = new_node();
57            cur->line = line;
58            return;
59        }
60
61        T mid = l + (r - l) / 2;
62        if (r - l <= eps) return;
63
64        // if max lichao, change to >
65        if (line.val(mid) < cur->line.val(mid))
66            swap(line, cur->line);
67
68        // if max lichao, change to >
69        if (line.val(l) < cur->line.val(l)) insert(cur->lc,
70            l, mid, line);
71        else insert(cur->rc, mid + 1, r, line);
72    }
73
74    T query(Node* &cur, T l, T r, T x) {
75        if (!cur) return identity;
76
77        T mid = l + (r - l) / 2;
78        T res = cur->line.val(x);
79        if (r - l <= eps) return res;
80
81        // if max lichao, change min to max
82        if (x <= mid) return min(res, query(cur->lc, l, mid
83            , x));
84        else return min(res, query(cur->rc, mid + 1, r, x));
85    }
86
87    void insert(T m, T c) { insert(root, L, R, Line(m, c
88        )); }
89
90    T query(T x) { return query(root, L, R, x); }

```

3.9 LIS - Longest Increasing Subsequence

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

```

1 int lis(vi &a){
2     int n = a.size();
3     vi len(n+1, INF);
4     len[0] = -INF;
5     for (int i = 0; i < n; i++){
6         int l = upper_bound(len.begin(), len.end(), a[i
7             ]) - len.begin();
8         if(len[l-1] < a[i] && a[i] < len[l]) len[l] = a
9             [i];
10
11        int ans = 0;
12        for (int i = 0; i <= n; i++){
13            if (len[i] < INF) ans = i;
14        }
15    }

```

3.10 SOSDP

```

1 int k; // amount of bits
2 vi a(1<<k);
3 // sosdp
4 for (int bit = 0; bit < k; bit++){
5     for (int mask = 0; mask < (1<<k); mask++){
6         if ((1<<bit) & mask) {
7             a[mask] += a[mask ^ (1<<bit)];
8         }
9     }
10 }
11
12 // do stuff (such as multiplication for OR convolution)

```

```

13 // sosdp inverse
14 for (int bit = 0; bit < k; bit++){
15     for (int mask = 0; mask < (1<<k); mask++){
16         if ((1<<bit) & mask) {
17             a[mask] -= a[mask ^ (1<<bit)];
18         }
19     }
20 }
21

```

3.11 Subset Sum

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

```

1 vvi dp(n+1, vi(sum+1));
2 vvii p(n+1, vii(sum+1));
3
4 dp[0][0] = 1;
5
6 for (int i = 1; i <= n; i++){
7     for (int s = 1; s <= sum; s++){
8         if (s-a[i-1] >= 0 && dp[i-1][s-a[i-1]]) {
9             // sum is possible taking item i
10            p[i][s] = {i-1, s-a[i-1]};
11            dp[i][s] = 1;
12        } else if (dp[i-1][s]) {
13            // sum not possible taking item i
14            // but still possible with other items (<i)
15            p[i][s] = {i-1, s};
16            dp[i][s] = 1;
17        }
18    }
19 }
20
21 if (!dp[n][target]) {
22     cout << -1 << endl;
23     return;
24 }
25
26 vi subset;
27 ii pos = fn(target);
28 while(pos != ii(0,0)){
29     auto [i,s] = pos;
30     if (p[i][s].second != s) subset.push_back(a[i-1]);
31     pos = p[i][s];
32 }

```

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: $\mathcal{O}(n)$

```

1 vvi adj;
2 vi sum_going_down, sum_going_up, sz;
3

```

```

4 void dfs(int u, int p){
5     for (auto v : adj[u]){
6         if (v == p) continue;
7         dfs(v, u);
8         sz[u] += sz[v];
9         sum_going_down[u] += sum_going_down[v];
10    }
11    sum_going_down[u] += sz[u];
12 }
13
14 void dfs2(int u, int p, int par_ans){
15     int up_amount = sz[0] - sz[u];
16     sum_going_up[u] += par_ans + up_amount;
17     int sum = sum_going_down[u];
18     for (auto v : adj[u]){
19         if (v == p) continue;
20         int par_amount = sz[0] - sz[v];
21         dfs2(v, u, par_ans + par_amount + sum - (
22             sum_going_down[v]+sz[v]));
23    }
24 }
25
26 void solve(){
27     int n; cin >> n;
28     adj = vvi(n);
29     sum_going_down = sum_going_up = vi(n);
30     sz = vi(n,1);
31
32     for (int i = 1; i < n; i++){
33         int a, b; cin >> a >> b;
34         a--; b--;
35         adj[a].push_back(b);
36         adj[b].push_back(a);
37     }
38
39     dfs(0,0);
40     dfs2(0,0,0);
41
42     for (int i = 0; i < n; i++){
43         cout << sum_going_down[i]+sum_going_up[i] << ',';
44     }
45     cout << endl;
46 }

```

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

```

1 struct EdgeHLD {
2     int n, timer = 0;
3     vvi adj;
4     vi parent, depth, size, heavy, head, value;
5     vi tin, tout;
6
7     segtree seg;
8
9     void init(int _n, vvii& _adj) {
10        n = _n;
11        adj = _adj;
12        value.assign(n, 0);
13        parent.assign(n, -1);
14        depth.assign(n, 0);
15        size.assign(n, 0);
16    }

```

```

17     heavy.assign(n, -1);
18     head.assign(n, 0);
19     tin.assign(n, 0);
20     tout.assign(n, 0);
21     timer = 0;
22
23     // edgeWeight[v] = weight of edge (parent[v], v),
24     // for v>0
25     // root (0) has no parent, so its value is dummy
26     // (0)
27     dfs1(0,0,0);
28     dfs2(0, 0);
29
30     vi linear(n);
31     for (int u = 0; u < n; u++)
32         linear[tin[u]] = value[u]; // position stores
33         edge weight
34     seg.init(linear);
35
36     int dfs1(int u, int p, int w) {
37         size[u] = 1;
38         parent[u] = p;
39         value[u] = w;
40         int max_sz = 0;
41         for (auto [v,w] : adj[u]) {
42             if (v == p) continue;
43             depth[v] = depth[u] + 1;
44             int sz = dfs1(v, u, w);
45             size[u] += sz;
46             if (sz > max_sz) {
47                 max_sz = sz;
48                 heavy[u] = v;
49             }
50         }
51         return size[u];
52     }
53
54     void dfs2(int u, int h) {
55         tin[u] = timer++;
56         head[u] = h;
57         if (heavy[u] != -1)
58             dfs2(heavy[u], h);
59         for (auto [v,w] : adj[u]) {
60             if (v != parent[u] && v != heavy[u])
61                 dfs2(v, v);
62         }
63         tout[u] = timer;
64     }
65
66     // u deve ser o filho
67     void update_edge(int u, int val) {
68         seg.set(tin[u], val);
69     }
70
71     void rangeUpdate(int u, int v, int x) {
72         while (head[u] != head[v]) {
73             if (depth[head[u]] < depth[head[v]]) swap(u, v);
74             seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
75             u = parent[head[u]];
76         }
77         if (depth[u] > depth[v]) swap(u, v);
78         seg.rangeUpdate(tin[u] + 1, tin[v] + 1, x); // +1
79         to skip LCA's edge
80     }
81
82     void update_subtree(int u, int x) {
83         // updates all edges in subtree of u (skip incoming

```

```

84         edge to u)
85         seg.rangeUpdate(tin[u] + 1, tout[u], x);
86     }
87
88     segtree::node query(int u, int v) {
89         segtree::node res = seg.NEUTRAL;
90         while (head[u] != head[v]) {
91             if (depth[head[u]] < depth[head[v]]) swap(u, v);
92             res = seg.merge(res, seg.query(tin[head[u]], tin[
93                 u] + 1));
94             u = parent[head[u]];
95         }
96         if (depth[u] > depth[v]) swap(u, v);
97         res = seg.merge(res, seg.query(tin[u] + 1, tin[v] +
98             1)); // skip LCA's edge
99         return res;
100    }
101
102 }

```

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.

```

1 struct HLD {
2     int n, timer = 0;
3     vvi adj;
4     vi parent, depth, size, heavy, head, value;
5     vi tin, tout;
6
7     segtree seg;
8
9     void init(int _n, vi& _value, vvi& _adj) {
10        n = _n;
11        adj = _adj;
12        value = _value;
13        parent.assign(n, -1);
14        depth.assign(n, 0);
15        size.assign(n, 0);
16        heavy.assign(n, -1);
17        head.assign(n, 0);
18        tin.assign(n, 0);
19        tout.assign(n, 0);
20        timer = 0;
21
22        dfs1(0);
23        dfs2(0, 0);
24
25        vi linear(n);
26        for (int u = 0; u < n; u++)
27            linear[tin[u]] = value[u];

```

```

28     seg.init(linear);
29 }
30 }
31 int dfs1(int u) {
32     size[u] = 1;
33     int max_sz = 0;
34     for (int v : adj[u]) {
35         if (v == parent[u]) continue;
36         parent[v] = u;
37         depth[v] = depth[u] + 1;
38         int sz = dfs1(v);
39         size[u] += sz;
40         if (sz > max_sz) {
41             max_sz = sz;
42             heavy[u] = v;
43         }
44     }
45     return size[u];
46 }
47 void dfs2(int u, int h) {
48     tin[u] = timer++;
49     head[u] = h;
50     if (heavy[u] != -1)
51         dfs2(heavy[u], h);
52     for (int v : adj[u]) {
53         if (v != parent[u] && v != heavy[u])
54             dfs2(v, v);
55     }
56     tout[u] = timer;
57 }
58 void update(int u, int val) {
59     seg.set(tin[u], val);
60 }
61 void rangeUpdate(int u, int v, int x) {
62     while (head[u] != head[v]) {
63         if (depth[head[u]] < depth[head[v]]) swap(u, v);
64         seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
65         u = parent[head[u]];
66     }
67     if (depth[u] > depth[v]) swap(u, v);
68     seg.rangeUpdate(tin[u], tin[v] + 1, x);
69 }
70 void update_subtree(int u, int x) {
71     seg.rangeUpdate(tin[u], tout[u], x);
72 }
73 segtree::node query(int u, int v) {
74     segtree::node res = seg.NEUTRAL;
75     while (head[u] != head[v]) {
76         if (depth[head[u]] < depth[head[v]])
77             swap(u, v);
78         res = seg.merge(res, seg.query(tin[head[u]], tin[u]+1));
79         u = parent[head[u]];
80     }
81     if (depth[u] > depth[v]) swap(u, v);
82     res = seg.merge(res, seg.query(tin[u], tin[v]+1));
83     return res;
84 }
85 segtree::node query_subtree(int u){
86     return seg.query(tin[u], tout[u]);
87 }
88 }
89 
```

4.4 LCA - RMQ

Pre-computation: $\mathcal{O}(n \log n)$ Queries: $\mathcal{O}(1)$ OBS: call first $dfs(root)$ and then $buildSparseTable()$ before making queries. Also remember to call $eulertournodes.reserve(2 \cdot n)$ and $eulertourdepths.reserve(2 \cdot n)$ to optimize memory allocation time of $pushback$.

```

1 int n, timer = 0;
2 vi tin, dep, et_nodes, et_depths;
3 vvi ch;
4 vvii sparse_table;
5
6 void dfs(int u) {
7     et_nodes.push_back(u);
8     et_depths.push_back(dep[u]);
9     tin[u] = timer++;
10
11    for (int v : ch[u]) {
12        dep[v] = dep[u] + 1;
13        dfs(v);
14        et_nodes.push_back(u);
15        et_depths.push_back(dep[u]);
16    }
17
18    timer++;
19 }
20 void buildSparseTable() {
21     int m = et_depths.size();
22     sparse_table.assign(LOGN, vii(m));
23
24     for (int i = 0; i < m; i++) {
25         sparse_table[0][i] = {et_depths[i], i};
26     }
27
28     for (int i = 1; (1 << i) <= m; i++) {
29         int len = 1 << i;
30         for (int time = 0; time + len <= m; time++) {
31             ii ans1 = sparse_table[i-1][time];
32             ii ans2 = sparse_table[i-1][time + len/2];
33             sparse_table[i][time] = min(ans1, ans2);
34         }
35     }
36 }
37 }
38 // TODO: change to struct sparse table for RMQ
39
40 int lca(int u, int v) {
41     int tu = tin[u];
42     int tv = tin[v];
43     if (tu > tv) swap(tu, tv);
44
45     int k = __bit_width((tv - tu + 1)) - 1;
46
47     ii ans1 = sparse_table[k][tu];
48     ii ans2 = sparse_table[k][tv - (1 << k) + 1];
49
50     if (ans1.first <= ans2.first) {
51         return et_nodes[ans1.second];
52     }
53     return et_nodes[ans2.second];
54 }
55 }
```

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.

```

1 vvi adj, up;
2 vi tin, tout;
3 int timer = 0;
4
5 void dfs(int u, int p){
6     tin[u] = timer++;
7     for (auto v : adj[u]){
8         if (v == p) continue;
9         up[v][0] = u;
10        for (int dist = 1; dist < LOGN; dist++){
11            up[v][dist] = up[up[v][dist-1]][dist-1];
12        }
13    }
14    dfs(v);
15    tout[u] = timer++;
16 }
17
18 int isAncestor(int u, int v){
19     return tin[u] <= tin[v] && tout[v] <= tout[u];
20 }
21
22 int lca(int u, int v){
23     if (isAncestor(u,v)) return u;
24     if (isAncestor(v,u)) return v;
25     for (int dist = LOGN-1; dist >= 0; dist--){
26         if (!isAncestor(up[u][dist],v)) u = up[u][dist];
27     }
28 }
29 }
```

5 Problemas clássicos

5.1 2SAT

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

```

1 // para adicionar negacao usar ~u
2 // Ex: a clausula (a v ~b) se traduz para add_or(a, ~b)
3 struct TwoSatSolver {
4     int n;
5     vvi adj, adjT;
6     vector<bool> vis, assignment;
7     vi topo, scc;
8
9     void build(int _n){
10        n = 2*_n;
11        adj.assign(n,vi());
12        adjT.assign(n,vi());
13    }
14
15    int get(int u){
16        if (u < 0) return 2*(~u)+1;
17        else return 2*u;
18    }
19
20    // u -> v
21    void add_impl(int u, int v){
22        u = get(u), v = get(v);
23        adj[u].push_back(v);
24    }
25 }
```

```

24     adjT[v].push_back(u);
25     adj[v^1].push_back(u^1);
26     adjT[u^1].push_back(v^1);
27 }
28
29 // u // v
30 void add_or(int u, int v){
31     add_impl(~u, v);
32 }
33
34 // u && v
35 void add_and(int u, int v){
36     add_or(u,u); add_or(v,v);
37 }
38
39 // u ^ v (equiv of x != v)
40 void add_xor(int u, int v){
41     add_impl(u, ~v);
42     add_impl(~u, v);
43 }
44
45 // u == v
46 void add_equals(int u, int v){
47     add_impl(u, v);
48     add_impl(v, u);
49 }
50
51 void toposort(int u){
52     vis[u] = true;
53     for (int v : adj[u])
54         if (!vis[v]) toposort(v);
55     topo.push_back(u);
56 }
57
58 void dfs(int u, int c){
59     scc[u] = c;
60     for (int v : adjT[u])
61         if (!scc[v]) dfs(v,c);
62 }
63
64 pair<bool, vector<bool>> solve(){
65     topo.clear();
66     vis.assign(n, false);
67
68     for (int i = 0; i < n; i++)
69         if (!vis[i]) toposort(i);
70
71     reverse(topo.begin(), topo.end());
72
73     scc.assign(n, 0);
74     int c = 0;
75     for (int u : topo)
76         if (!scc[u]) dfs(u,++c);
77
78     assignment.assign(n/2, false);
79     for (int i = 0; i < n; i += 2){
80         if (scc[i] == scc[i+1]) return {false, {}};
81         assignment[i/2] = scc[i] > scc[i+1];
82     }
83
84     return {true, assignment};
85 }
86 
```

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.

```

1 vi next_greater_elem(n, n);
2
3 stack<ii> st;
4 for (int i = 0; i < n; i++){
5     while (!st.empty() && st.top().first < h[i]){
6         next_greater_elem[st.top().second] = i;
7         st.pop();
8     }
9     st.emplace(h[i], i);
10 } 
```

6 Strings

6.1 Hashing

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

```

1 class Hashing{
2     const int mod0 = 1e9+7;
3     vi pmod0;
4     vull pmod1;
5
6     public:
7         void CalcP(int mn, int n){
8             random_device rd;
9             uniform_int_distribution<int> dist(mn+2, mod0
10                         -1);
11            int p = dist(rd);
12            if(p % 2 == 0) p--;
13            pmod0 = vi(n);
14            pmod1 = vull(n);
15            pmod0[0] = pmod1[0] = 1;
16            for(int i = 1; i < n; i++){
17                pmod0[i] = (pmod0[i-1] * p) % mod0;
18                pmod1[i] = (pmod1[i-1] * p);
19            }
20        }
21        viull DistinctSubstrHashes(string base, int
22                                     offsetVal){
23            int n = base.size();
24            viull ans;
25            for(int i = 0; i < n; i++){
26                int h0 = 0;
27                ull h1 = 0;
28                for(int j = i; j < n; j++){
29                    h0 = (h0 + (base[j]-offsetVal)*pmod0[j-
30                                i]) % mod0;
31                    h1 = (h1 + (base[j]-offsetVal)*pmod1[j-
32                                i]);
33                    ans.push_back(iull(h0, h1));
34                }
35            }
36            sort(ans.begin(), ans.end());
37            auto last = unique(ans.begin(), ans.end());
38            ans.erase(last, ans.end());
39            return ans;
40        }
41        viull WindowHash(string data, int offsetVal, int
42                          lenWindow){
43            int n = data.size();
44            int h0 = 0;
45            ull h1 = 0;
46            viull ans; 
```

```

44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59 };
```

6.2 KMP

```

1 vi compute_lps(const string &pat){
2     int m = pat.length();
3     vi lps(m);
4     int len = 0;
5     for (int i = 1; i < m; i++){
6         while(len > 0 && pat[i] != pat[len])
7             len = lps[len-1];
8         if (pat[i] == pat[len]) len++;
9         lps[i] = len;
10    }
11    return lps;
12 }
13
14 // find all occurrences
15 vi kmp_search(const string &txt, const string &pat){
16     int n = txt.length();
17     int m = pat.length();
18     if (m == 0) return {};
19     vi lps = compute_lps(pat);
20     vi occurrences;
21     int j = 0;
22     for (int i = 0; i < n; i++){
23         while (j > 0 && txt[i] != pat[j])
24             j = lps[j-1];
25         if (txt[i] == pat[j]) j++;
26         if (j == m) {
27             occurrences.push_back(i-m+1);
28             j = lps[j-1];
29         }
30     }
31     return occurrences;
32 }
33
34 // find all occurrences (simpler version)
35 vi kmp_search(const string &txt, const string &pat){
36     int n = txt.length(), m = pat.length();
37     vi lps = compute_lps(pat + '#' + txt);
38     vi occurrences;
39     for (int i = 0; i < n+m+1; i++){
40         if (lps[i] == pat.length())
41             occurrences.push_back(i-m*2); 
```

```

43     }
44     return occurrences;
45 }
46
47 // borda sao os prefixos que tambem sao sufixos
48 vi find_borders(const string &s){
49     vi lps = compute_lps(s);
50     int i = s.length()-1;
51
52     vi ans;
53     while (lps[i] > 0){
54         ans.push_back(lps[i]);
55         i = lps[i]-1;
56     }
57     reverse(ans.begin(), ans.end());
58     return ans;
59 }

```

6.3 Suffix Array

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

```

1 struct SuffixArray {
2     int sz;
3     vi suff_ind, lcp;
4     viii suffs;
5
6     void radix_sort() {
7         if (sz <= 1) return;
8         viii suffs_new(sz);
9         vi cnt(sz + 1, 0); /*rever esse tamanho*/
10
11        for (auto& item : suffs) cnt[item.first.second]
12            ]++;
13        for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
14        for (int i = sz - 1; i >= 0; --i) suffs_new[--cnt[suffs[i].first.second]] = suffs[i];
15
16        cnt.assign(sz + 1, 0);
17        for (auto& item : suffs_new) cnt[item.first.
18            first]++;
19        for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
20        for (int i = sz - 1; i >= 0; --i) suffs_new[-cnt[
21            suffs_new[i].first.first]] = suffs_new[i];
22    }
23
24    void build_lcp(vi& a) {
25        lcp.assign(sz, 0);
26        vi rank(sz);
27        for (int i = 0; i < sz; ++i) rank[suff_ind[i]] = i;
28
29        int h = 0;
30        for (int i = 0; i < sz; ++i) {
31            if (rank[i] == sz - 1) { h = 0; continue; }
32            if (h > 0) h--;
33            int j = suff_ind[rank[i] + 1];
34            while (i + h < sz && j + h < sz && a[i + h]
35                == a[j + h]) h++;
36            lcp[rank[i] + 1] = h;
37        }
38
39    void build(vi& a) {
40        a.push_back(0);
41        sz = a.size();
42        suffs.resize(sz);

```

```

43
44        suff_ind.resize(sz);
45        vi equiv(sz);
46
47        for (int i = 0; i < sz; ++i) suffs[i] = iii(ii(
48            a[i], a[i]), i);
49        radix_sort();
50        for (int i = 1; i < sz; ++i) {
51            auto [c, ci] = suffs[i];
52            auto [p, pi] = suffs[i-1];
53            equiv[ci] = equiv[pi] + (c > p);
54        }
55
56        for (int suflen = 1; suflen < sz; suflen *= 2)
57            {
58                for (int i = 0; i < sz; ++i) {
59                    suffs[i] = {{equiv[i], equiv[(i +
60                        suflen) % sz]}, i};
61                }
62                radix_sort();
63                for (int i = 1; i < sz; ++i) {
64                    auto [c, ci] = suffs[i];
65                    auto [p, pi] = suffs[i-1];
66                    equiv[ci] = equiv[pi] + (c > p);
67                }
68
69                for(int i = 0; i < sz; ++i) suff_ind[i] = suffs
70                    [i].second;
71                build_lcp(a);
72
73                a.pop_back();
74                sz--;
75                suff_ind.erase(suff_ind.begin());
76                lcp.erase(lcp.begin());
77            }
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97

```

6.4 Suffix Automaton

```

1 struct SAM {
2     struct State {
3         int len, link;
4         ll cnt = 0;
5         int first_occ=-1;
6         map<char,int> next;
7     };
8
9     vector<State> st;
10    int last;
11
12    SAM(string s){
13        st.push_back({0,-1,0,-1});
14        last = 0;
15        for (int i = 0; i < s.length(); i++){
16            extend(s[i],i);
17        }
18        calc_cnt();
19    }
20
21    void extend(char c, int id){
22        int cur = st.size();
23        st.push_back({st[last].len+1,0,1,id});
24        int p = last;
25        while(p!= -1 && st[p].next.count(c)==0){
26            st[p].next[c] = cur;
27            p = st[p].link;
28        }
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97

```

```

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);

```

```

98     for (int i = 1; i < st.size(); i++) {
99         int l = st[st[i].link].len+1;
100        int r = st[i].len;
101        diff[l]++;
102        diff[r+1]--;
103    }
104    vi ans(n+1);
105    ans[0] = diff[0];
106    for (int i = 1; i <= n; i++) {
107        ans[i] = ans[i-1]+diff[i];
108    }
109
110    return ans;
111}
112
113 void calc_paths(int u){
114     if (dp[u] != -1) return;
115     dp[u]=1;
116     for (auto [c,v] : st[u].next){
117         calc_paths(v);
118         dp[u] += dp[v];
119     }
120 }
121
122 string find_kth(int k){
123     dp.assign(st.size(), -1);
124     calc_paths(0);
125     int u = 0;
126     string ans = "";
127     while(k>0){
128         for (auto [c,v] : st[u].next){
129             bool ok = false;
130             if (k <= dp[v]){
131                 ans += c;
132                 u = v;
133                 k--;
134                 ok = true;
135                 break;
136             }
137             if (!ok) k-=dp[v];
138         }
139     }
140     return ans;
141 }
142
143 void calc_paths_with_repetitions(int u){
144     if (dp[u] != -1) return;
145     dp[u]=st[u].cnt;
146     for (auto [c,v] : st[u].next){
147         calc_paths_with_repetitions(v);
148         dp[u] += dp[v];
149     }
150 }
151
152 string find_kth_with_repetitions(int k){
153     dp.assign(st.size(), -1);
154     calc_paths_with_repetitions(0);
155     int u = 0;
156     string ans = "";
157     while(k>0){
158         for (auto [c,v] : st[u].next){
159             bool ok = false;
160             if (k <= dp[v]){
161                 ans += c;
162                 k-=st[v].cnt;
163                 u = v;
164                 ok = true;
165                 break;
166             }
167             if (!ok) k-=dp[v];
168         }
169     }
170     return ans;
171 }
172 }

```

```

168         }
169     }
170     return ans;
171 }
172 }

```

6.5 Z

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

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

```

1 vi compute_z(const string &s) {
2     int n = s.length();
3     vi z(n);
4     int l = 0, r = 0;
5
6     for (int i = 1; i < n; i++) {
7         if (i <= r)
8             z[i] = min(r - i + 1, z[i-1]);
9
10        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
11            z[i]++;
12        if (i + z[i] - 1 > r) {
13            l = i;
14            r = i + z[i] - 1;
15        }
16    }
17
18    return z;
19 }
20
21 vi find_occurrences(const string &txt, const string &
22 pat){
23     vi occurrences;
24     vi z = compute_z(pat + '#' + txt);
25     int n = txt.length(), m = pat.length();
26     for (int i = 0; i < n+m+1; i++){
27         if (z[i] == m) occurrences.push_back(i-m-1);
28     }
29     return occurrences;

```

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

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

- 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.

```

1 // n escolhe k
2 // linha n, coluna k no triangulo (indexadas em 0)
3 int pascal(int n, int k){
4     int num = fat[n];
5     int den = (fat[k]*fat[n-k])%ZAP;
6     return (num*expbin(den, ZAP-2))%ZAP;
7 }
```

7.2 Convolutions

7.2.1 AND convolution

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

```

1 vector<mint<MOD>> and_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = a.size(); // must be pow of 2
4     for (int j = 1; j < n; j <= 1) {
5         for (int i = 0; i < n; i++) {
6             if (i&j) {
7                 a[i^j] += a[i];
8                 b[i^j] += b[i];
9             }
10        }
11    }
12    for (int i = 0; i < n; i++) a[i] *= b[i];
13    for (int j = 1; j < n; j <= 1) {
14        for (int i = 0; i < n; i++) {
15            if (i&j) a[i^j] -= a[i];
16        }
17    }
18
19
20 }
```

7.2.2 GCD convolution

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

```

1 vector<mint<MOD>> gcd_conv(vi a, vi b){
2     int n = (int)max(a.size(), b.size());
3     a.resize(n);
4     b.resize(n);
5     vector<mint<MOD>> c(n);
6     for (int i = 1; i < n; i++) {
7         mint<MOD> x = 0;
8         mint<MOD> y = 0;
9         for (int j = i; j < n; j += i) {
10            x += a[j];
11            y += b[j];
12        }
13        c[i] = x*y;
14    }
15    for (int i = n-1; i >= 1; i--)
16        for (int j = 2 * i; j < n; j += i)
17            c[i] -= c[j];
18
19    return c;
20 }
```

7.2.3 LCM convolution

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

```

1 vector<mint<MOD>> lcm_conv(vi a, vi b){
2     int n = (int)max(a.size(), b.size());
3     a.resize(n);
4     b.resize(n);
5     vector<mint<MOD>> c(n), x(n), y(n);
6     for (int i = 1; i < n; i++) {
7         for (int j = i; j < n; j += i) {
8             x[j] += a[i];
9             y[j] += b[i];
10        }
11        c[i] = x[i]*y[i];
12    }
13    for (int i = 1; i < n; i++)
14        for (int j = 2 * i; j < n; j += i)
15            c[j] -= c[i];
16
17    return c;
18 }
```

7.2.4 OR convolution

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

```

1 vector<mint<MOD>> or_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = a.size(); // must be pow of 2
4     for (int j = 1; j < n; j <= 1) {
5         for (int i = 0; i < n; i++) {
6             if (i&j) {
7                 a[i] += a[i^j];
8                 b[i] += b[i^j];
9             }
10        }
11    }
12 }
```

```

9     }
10
11     for (int i = 0; i < n; i++) a[i] *= b[i];
12
13     for (int j = 1; j < n; j <= 1) {
14         for (int i = 0; i < n; i++) {
15             if (i&j) a[i] -= a[i^j];
16         }
17     }
18 }
19
20 return a;
21 }
```

7.2.5 XOR convolution

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

```

1 void fwht(vector<mint<MOD>> &a, bool inv){
2     int n = a.size(); // must be pow of 2
3     for (int step = 1; step < n; step <= 1) {
4         for (int i = 0; i < n; i += 2*step) {
5             for (int j = i; j < i+step; j++) {
6                 auto u = a[j];
7                 auto v = a[j+step];
8                 a[j] = u+v;
9                 a[j+step] = u-v;
10            }
11        }
12    }
13    if (inv) for (auto &x : a) x /= n;
14
15    vector<mint<MOD>> xor_conv(vector<mint<MOD>> a, vector<
16        mint<MOD>> b){
17        int n = a.size();
18        fwht(a, 0), fwht(b, 0);
19        for (int i = 0; i < n; i++) a[i] *= b[i];
20        fwht(a, 1);
21        return a;
22    }
23 }
```

7.3 Extended Euclid

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

```

1 int extended_gcd(int a, int b, int &x, int &y) {
2     x = 1, y = 0;
3     int x1 = 0, y1 = 1;
4     while (b) {
5         int q = a / b;
6         tie(x, x1) = make_tuple(x1, x - q * x1);
7         tie(y, y1) = make_tuple(y1, y - q * y1);
8         tie(a, b) = make_tuple(b, a - q * b);
9     }
10    return a;
11 }
```

7.4 Factorization

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

```

1 // OBS: tem outras variantes mais rápidas no caderno da
2 // UDESC
3 // O(sqrt(n)) fatores repetidos
4 vi fatora(int n) {
5     vi factors;
6     for (int x = 2; x * x <= n; x++) {
7         while (n % x == 0) {
8             factors.push_back(x);
9             n /= x;
10        }
11    }
12    if (n > 1) factors.push_back(n);
13    return factors;
14 }
15
16 // O(sqrt(n))
17 // Calcula a quantidade de divisores de um numero n.
18 int qtdDivisores(int n) {
19     int ans = 1;
20     for (int i = 2; i * i <= n; i += 2) {
21         int exp = 0;
22         while (n % i == 0) {
23             n /= i; exp++;
24         }
25         if (exp > 0) ans *= (exp + 1);
26         if (i == 2) i--;
27     }
28     if (n > 1) ans *= 2;
29     return ans;
30 }
31
32 // O(sqrt(n))
33 // Calcula a soma de todos os divisores de um numero n.
34 int somaDivisores(int n) {
35     int ans = 1;
36     for (int i = 2; i * i <= n; i += 2) {
37         if (n % i == 0) {
38             int exp = 0;
39             while (n % i == 0) {
40                 n /= i; exp++;
41             }
42             int aux = expbin(i, exp + 1);
43             ans *= ((aux - 1) / (i - 1));
44         }
45         if (i == 2) i--;
46     }
47
48     if (n > 1) ans *= (n + 1);
49     return ans;
50 }
```

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: $\mathcal{O}(n)$

```

1 void fft(vector<cd> &a, bool invert){
2     int len = a.size();
3     for (int i = 1, j = 0; i < len; i++) {
4         int bit = len / 2;
5         while (j >= bit) {
6             j -= bit;
7             bit /= 2;
8         }
9         if (i > j) swap(a[i], a[j]);
10    }
11 }
```

```

4     int bit = len >> 1;
5     while(bit & j){
6         j ^= bit;
7         bit >>= 1;
8     }
9     j ^= bit;
10    if(i < j) swap(a[i], a[j]);
11
12    for(int l = 2; l <= len; l <= 1){
13        double ang = 2*PI/l * (invert ? -1: 1);
14        cd wd(cos(ang), sin(ang));
15        for(int i = 0; i < len; i += 1){
16            cd w(1);
17            for(int j = 0; j < l/2; j++){
18                cd u = a[i+j], v = a[i+j+1/2];
19                a[i+j] = u+w*v;
20                a[i+j+1/2] = u-w*v;
21                w *= wd;
22            }
23        }
24    }
25    if(invert){
26        for(int i = 0; i < len; i++) {
27            a[i] /= len;
28        }
29    }
30 }

```

7.6 Inclusion-Exclusion Principle

TODO: rewrite math statement

```

1 // Exemplo:
2 // Contar numeros de 1 a n divisiveis por uma lista de
3 // primos.
4 vi primes;
5 int factors = primes.size();
6 int total_divisible = 0;
7
8 // Itera pelas bitmaskas nao vazias de 'primes'
9 for (int i = 1; i < (1 << factors); i++) {
10    int current_lcm = 1;
11    int subset_size = 0;
12
13    // calcula lcm do subconjunto
14    for (int j = 0; j < factors; j++) {
15        if (i & (1<<j)) {
16            subset_size++;
17            current_lcm = lcm(current_lcm, primes[j]);
18            if (current_lcm > n) break;
19        }
20    }
21
22    if (current_lcm > n) {
23        continue;
24    }
25
26    int count = n / current_lcm;
27
28    // Aplica o Princípio da Inclusão-Exclusão:
29    // Se o tamanho do subconjunto eh ímpar, adiciona.
30    // Se o tamanho do subconjunto eh par, subtrai.
31    if (subset_size & 1) {
32        total_divisible += count;
33    } else {
34        total_divisible -= count;
35    }
36 }

```

36 }

7.7 Mint

```

1 template<ll MOD>
2 struct mint {
3     ll val;
4     mint(ll v = 0) {
5         if (v < 0) v = v % MOD + MOD;
6         if (v >= MOD) v %= MOD;
7         val = v;
8     }
9     mint& operator+=(const mint& other) {
10        val += other.val;
11        if (val >= MOD) val -= MOD;
12        return *this;
13    }
14    mint& operator-=(const mint& other) {
15        val -= other.val;
16        if (val < 0) val += MOD;
17        return *this;
18    }
19    mint& operator*=(const mint& other) {
20        val = (val * other.val) % MOD;
21        return *this;
22    }
23    mint& operator/=(const mint& other) {
24        val = (val * inv(other).val) % MOD;
25        return *this;
26    }
27    friend mint operator+(mint a, const mint& b) {
28        return a += b;
29    }
30    friend mint operator-(mint a, const mint& b) {
31        return a -= b;
32    }
33    friend mint operator*(mint a, const mint& b) {
34        return a *= b;
35    }
36    friend mint operator/(mint a, const mint& b) {
37        return a /= b;
38    }
39    static mint power(mint b, ll e) {
40        mint ans = 1;
41        while (e > 0) {
42            if (e & 1) ans *= b;
43            b *= b;
44            e /= 2;
45        }
46        return ans;
47    }
48    static mint inv(mint n) { return power(n, MOD - 2);
49    }
50 };

```

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

```

1 int modInverse(int a, int m) {
2     int x, y;
3     int g = extendedGcd(a, m, x, y);
4     if (g != 1) return -1;

```

```

5     return (x % m + m) % m;
6 }

```

7.9 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.

7.9.1 NTT-Friendly Primes and Roots

NTT-friendly primes and their primitive roots:

- Modulus: 998244353, Primitive Root: 3, Maximum N: 2^{23}
- Modulus: 734003201, Primitive Root: 3, Maximum N: 2^{20}
- Modulus: 167772161, Primitive Root: 3, Maximum N: 2^{25}
- Modulus: 469762049, Primitive Root: 3, Maximum 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).

```

1 template<typename T, ll MOD, ll ROOT>
2 void transform(vector<T>& a, bool invert) {
3     int n = a.size();
4
5     for (int i = 1, j = 0; i < n; i++) {
6         int bit = n >> 1;
7         for (; j & bit; bit >>= 1)
8             j ^= bit;
9         j ^= bit;
10        if (i < j) swap(a[i], a[j]);
11    }
12
13    for (int len = 2; len <= n; len <= 1) {
14        T wlen = T::power(ROOT, (MOD - 1) / len);
15        if (invert) wlen = T::inv(wlen);
16        for (int i = 0; i < n; i += len) {
17            T w = 1;

```

```

18     for (int j = 0; j < len / 2; j++) {
19         T u = a[i + j], v = a[i + j + len /
20             2] * w;
21         a[i + j] = u + v;
22         a[i + j + len / 2] = u - v;
23         w *= wlen;
24     }
25 }
26
27 if (invert) {
28     T n_inv = T::inv(n);
29     for (T& x : a)
30         x *= n_inv;
31 }
32
33 template<typename T, ll MOD, ll ROOT>
34 vector<ll> multiply(const vector<ll>& a, const
35     vector<ll>& b) {
36     vector<T> fa(a.begin(), a.end()), fb(b.begin(),
37         b.end());
38     int n = 1;
39     while (n < a.size() + b.size()) n <<= 1;
40     fa.resize(n);
41     fb.resize(n);
42
43     transform<T, MOD, ROOT>(fa, false);
44     transform<T, MOD, ROOT>(fb, false);
45
46     for (int i = 0; i < n; i++) fa[i] *= fb[i];
47
48     transform<T, MOD, ROOT>(fa, true);
49
50     vector<ll> result(n);
51     for (int i = 0; i < n; i++) result[i] = fa[i].val;
52     return result;
53 }
54
55 vector<ll> crt_multiply(const vector<ll>& a, const
56     vector<ll>& b) {
57     const ll mod1 = 998244353;
58     const ll root1 = 3;
59     using mint1 = mint<mod1>;
60     vector<ll> ans1 = NTT::multiply<mint1, mod1,
61         root1>(a, b);
62
63     const ll mod2 = 1004535809;
64     const ll root2 = 3;
65     using mint2 = mint<mod2>;
66     vector<ll> ans2 = NTT::multiply<mint2, mod2,
67         root2>(a, b);
68
69     int ans_size = a.size() + b.size() - 2;
70     ll M1_inv_M2 = mint<mod2>::inv(mod1).val;
71
72     vector<ll> final_result(ans_size + 1);
73     for (int i = 0; i <= ans_size; ++i) {
74         ll v1 = ans1[i];
75         ll v2 = ans2[i];
76         ll k = ((v2 - v1 + mod2) % mod2 * M1_inv_M2
77             ) % mod2;
78         final_result[i] = v1 + k * mod1;
79     }
80     return final_result;
81 }

```

7.10 Euler's Totient

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

```

1 int phi(int n){
2     int ans = n;
3     for (int i = 2; i*i <= n; i++){
4         if (n%i == 0){
5             while(n%i == 0) n/=i;
6             ans -= ans/i;
7         }
8     }
9     if (n>1) ans -= ans/n;
10    return ans;
11 }

```

8 Geometry

8.1 Convex hull - Graham Scan

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

```

1 #define CLOCKWISE -1
2 #define COUNTERCLOCKWISE 1
3 #define INCLUDE_COLLINEAR 0 // pode mudar
4
5 struct Point {
6     ll x, y;
7     bool operator==(Point const& t) const {
8         return x == t.x && y == t.y;
9     }
10 }
11
12 struct Vec {
13     int x, y, z;
14 };
15
16 Vec cross(Vec v1, Vec v2){
17     int x = v1.y*v2.z - v1.z*v2.y;
18     int y = -v1.x*v2.z + v1.z*v2.x;
19     int z = v1.x*v2.y - v1.y*v2.x;
20     return {x,y,z};
21 }
22
23 ll dist2(Point p1, Point p2){
24     int dx = p1.x-p2.x;
25     int dy = p1.y-p2.y;
26     return dx*dx+dy*dy;
27 }
28
29 ll orientation(Point pivot, Point a, Point b){
30     Vec va = {a.x-pivot.x, a.y-pivot.y, 0};
31     Vec vb = {b.x-pivot.x, b.y-pivot.y, 0};
32     Vec v = cross(va,vb);
33     if (v.z < 0) return CLOCKWISE;
34     if (v.z > 0) return COUNTERCLOCKWISE;
35     return 0;
36 }
37
38 bool clock_wise(Point pivot, Point a, Point b) {
39     int o = orientation(pivot, a, b);
40     return o < 0 || (INCLUDE_COLLINEAR && o == 0);
41 }

```

```

43 bool collinear(Point a, Point b, Point c) { return
44     orientation(a, b, c) == 0; }
45 vector<Point> convex_hull(vector<Point> &points, bool
46     counterClockwise) {
47     int n = points.size();
48     Point pivot = *min_element(points.begin(), points.
49         end(), [] (Point a, Point b) {
50         return ii(a.y, a.x) < ii(b.y, b.x);
51     });
52     sort(points.begin(), points.end(), [&] (Point a,
53         Point b) {
54         int o = orientation(pivot, a, b);
55         if (o == 0) return dist2(pivot, a) < dist2(
56             pivot, b);
57         return o == COUNTERCLOCKWISE;
58     });
59     if (INCLUDE_COLLINEAR) {
60         int i = n-1;
61         while (i >= 0 && collinear(pivot, points[i],
62             points.back())) i--;
63         reverse(points.begin() + i + 1, points.end());
64     }
65     vector<Point> hull;
66     for (auto p : points) {
67         while (hull.size() > 1 && !clock_wise(hull[hull.
68             size() - 2], hull.back(), p))
69             hull.pop_back();
70         hull.push_back(p);
71     }
72     if (!INCLUDE_COLLINEAR && hull.size() == 2 && hull
73         [0] == hull[1])
74         hull.pop_back();
75     if (counterClockwise && hull.size() > 1) {
76         vector<Point> reversed_hull = hull;
77         reverse(reversed_hull.begin() + 1,
78             reversed_hull.end());
79     }
80     return hull;
81 }

```

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 $|a - b| < \text{eps}!!!!$

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 2π
Time: $\mathcal{O}(n \log n)$

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 typedef long double ld;
4 #define eps 1e-9
5 #define pi 3.141592653589
6 #define int long long int
7
8
9 struct pt {
10     int x, y;
11     int operator==(pt b) {
12         return x == b.x && y == b.y;
13     }
14     int operator<(pt b) {
15         if(x == b.x) return y < b.y;
16         return x < b.x;
17     }
18     pt operator-(pt b) {
19         return {x - b.x, y - b.y};
20     }
21     pt operator+(pt b) {
22         return {x+b.x, y + b.y};
23     }
24 };
25 int cross(pt u, pt v) {
26     return u.x * v.y - u.y * v.x;
27 }
28 int dot(pt u, pt v) {
29     return u.x * v.x + u.y * v.y;
30 }
31 ld norm(pt u) {
32     return sqrt(dot(u, u));
33 }
34 ld dist(pt u, pt v) {
35     return norm(u - v);
36 }
37 int ccw(pt u, pt v) { // cuidado com colineares!!!!
38     return (cross(u, v) > eps)?1:(fabs(cross(u, v)) <
39         eps)?0:-1;
40 }
41 int pointInSegment(pt a, pt u, pt v) { // checks if a
42     lies in uv
43     if(ccw(v - u, a - u)) return 0;
44     vector<pt> pts = {a, u, v};
45     sort(pts.begin(), pts.end());
46     return pts[1] == a;
47 }
48 ld angle(pt u, pt v) { // angle between two vectors
49     ld c = cross(u, v);
50     ld d = dot(u, v);
51     return atan2l(c, d);
52 }
53 int intersect(pt sa, pt sb, pt ra, pt rb) { // not sure
54     if(it works when one of the segments is a point
55     pt s = sb - sa, r = rb - ra;
56     if(pointInSegment(sa, ra, rb) || pointInSegment(sb,
57         ra, rb) || pointInSegment(ra, sa, sb) ||
58         pointInSegment(rb, sa, sb)) return 1;
59 }
```

```

54     return !(ccw(s, ra - sa) == ccw(s, rb - sa) || ccw(
55         r, sa - ra) == ccw(r, sb - ra));
56 }
57 ld polygonArea(vector<pt>& p) {// not signed (for
58     signed area remove the absolute value at the end)
59     ld area = 0;
60     int n = p.size() - 1; // p[n] = p[0]
61     for(int i = 0; i < n; i++) {
62         area += cross(p[i], p[i + 1]);
63     }
64     return fabs(area)/2;
65 int pointInPolygon(pt a, vector<pt>& p) {// returns 0
66     for point in BOUNDARY, 1 for point in polygon and
67     -1 for outside
68     ld total = 0;
69     int n = p.size() - 1;
70     for(int i = 0; i < n; i++) {
71         pt u = p[i] - a;
72         pt v = p[i + 1] - a;
73         if(fabs(dist(p[i], a) + dist(p[i + 1], a) -
74             dist(p[i], p[i + 1])) < eps) {
75             return 0;
76         }
77         total += angle(u, v);
78     }
79
80
81 signed main() {
82     int n, m; scanf("%lld %lld", &n, &m);
83     vector<pt> p(n + 1);
84     for(int i = 0; i < n; i++) {
85         scanf("%lld %lld", &p[i].x, &p[i].y);
86     }
87     p[n] = p[0];
88
89     while(m--) {
90         pt a; scanf("%lld %lld", &a.x, &a.y);
91         int ans = pointInPolygon(a, p);
92         printf("%s\n", (ans > 0)? "INSIDE" : (ans? "OUTSIDE"
93             ": "BOUNDARY"));
94     }
95 }
```