

Columbia University: CU Later Team Reference Document

Kevin Yang, Innokentiy Kaurov, Eric Yuang Shao

May 21th 2024

Contents

Templates

Ken's template	1
Kevin's template	1
Kevin's Template Extended	1

Geometry

Strings

Manacher's algorithm	4
--------------------------------	---

Flows

$O(N^2M)$, on unit networks $O(N^{1/2}M)$	4
MCMF – maximize flow, then minimize its cost. $O(Fmn)$	4

Graphs

Kuhn's algorithm for bipartite matching	6
Hungarian algorithm for Assignment Problem . . .	6
Dijkstra's Algorithm	6
Eulerian Cycle DFS	6
SCC and 2-SAT	6
Finding Bridges	7
Virtual Tree	7
HLD on Edges DFS	7
Centroid Decomposition	7

Math

Binary exponentiation	8
Matrix Exponentiation: $O(n^3 \log b)$	8
Extended Euclidean Algorithm	8
Linear Sieve	8
Gaussian Elimination	8
NTT	9
FFT	9
is_prime	9
Berlekamp-Massey	10
Calculating k-th term of a linear recurrence	10

Data Structures

Fenwick Tree	11
Lazy Propagation SegTree	11
Sparse Table	11
Suffix Array and LCP array	12
Aho Corasick Trie	12
Convex Hull Trick	13
Li-Chao Segment Tree	13
Persistent Segment Tree	14

Miscellaneous

Ordered Set	14
Measuring Execution Time	14
Setting Fixed D.P. Precision	14
Common Bugs and General Advice	14

Templates

Ken's template

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define all(v) (v).begin(), (v).end()
4 typedef long long ll;
5 typedef long double ld;
6 #define pb push_back
7 #define sz(x) (int)(x).size()
8 #define fi first
9 #define se second
10 #define endl '\n'
```

Kevin's template

```
1 // paste Kaurov's Template, minus last line
2 typedef vector<int> vi;
3 typedef vector<ll> vll;
4 typedef pair<int, int> pii;
5 typedef pair<ll, ll> pll;
6 typedef pair<double, double> pdd;
7 const ld PI = acosl(-1);
8 const ll mod7 = 1e9 + 7;
9 const ll mod9 = 998244353;
10 const ll INF = 2*1024*1024*1023;
11 const char nl = '\n';
12 #define forn(i, n) for (int i = 0; i < int(n); i++)
13 ll k, n, m, u, v, w;
14 string s, t;
15
16 bool multiTest = 1;
17 void solve(int tt){
18 }
19
20 int main(){
21     ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
22     cout<<fixed<< setprecision(14);
23
24     int t = 1;
25     if (multiTest) cin >> t;
26     forn(ii, t) solve(ii);
27 }
```

Kevin's Template Extended

- to type after the start of the contest

```
1 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace __gnu_pbds;
5 template<class T> using ordered_set = tree<T, null_type,
6     ↪ less<T>, rb_tree_tag, tree_order_statistics_node_update>;
7 vi d4x = {1, 0, -1, 0};
8 vi d4y = {0, 1, 0, -1};
9 vi d8x = {1, 0, -1, 0, 1, 1, -1, -1};
10 vi d8y = {0, 1, 0, -1, 1, -1, 1, -1};
11 mt19937
12     ↪ rng(chrono::steady_clock::now().time_since_epoch().count());
```

Geometry

- Basic stuff

```
1 template<typename T>
2 struct TPoint{
3     T x, y;
4     int id;
5     static constexpr T eps = static_cast<T>(1e-9);
6     TPoint() : x(0), y(0), id(-1) {}
7     TPoint(const T& x_, const T& y_) : x(x_), y(y_), id(-1) {}
8     TPoint(const T& x_, const T& y_, const int id_) : x(x_),
9     ↪ y(y_), id(id_) {}
```

```

9
10 TPoint operator + (const TPoint& rhs) const {
11     return TPoint(x + rhs.x, y + rhs.y);
12 }
13 TPoint operator - (const TPoint& rhs) const {
14     return TPoint(x - rhs.x, y - rhs.y);
15 }
16 TPoint operator * (const T& rhs) const {
17     return TPoint(x * rhs, y * rhs);
18 }
19 TPoint operator / (const T& rhs) const {
20     return TPoint(x / rhs, y / rhs);
21 }
22 TPoint ort() const {
23     return TPoint(-y, x);
24 }
25 T abs2() const {
26     return x * x + y * y;
27 }
28 };
29 template<typename T>
30 bool operator< (TPoint<T>& A, TPoint<T>& B){
31     return make_pair(A.x, A.y) < make_pair(B.x, B.y);
32 }
33 template<typename T>
34 bool operator== (TPoint<T>& A, TPoint<T>& B){
35     return abs(A.x - B.x) <= TPoint<T>::eps && abs(A.y - B.y) <=
        ↪ TPoint<T>::eps;
36 }
37 template<typename T>
38 struct TLine{
39     T a, b, c;
40     TLine() : a(0), b(0), c(0) {}
41     TLine(const T& a_, const T& b_, const T& c_) : a(a_), b(b_),
        ↪ c(c_) {}
42     TLine(const TPoint<T>& p1, const TPoint<T>& p2){
43         a = p1.y - p2.y;
44         b = p2.x - p1.x;
45         c = -a * p1.x - b * p1.y;
46     }
47 };
48 template<typename T>
49 T det(const T& a11, const T& a12, const T& a21, const T& a22){
50     return a11 * a22 - a12 * a21;
51 }
52 template<typename T>
53 T sq(const T& a){
54     return a * a;
55 }
56 template<typename T>
57 T smul(const TPoint<T>& a, const TPoint<T>& b){
58     return a.x * b.x + a.y * b.y;
59 }
60 template<typename T>
61 T vmul(const TPoint<T>& a, const TPoint<T>& b){
62     return det(a.x, a.y, b.x, b.y);
63 }
64 template<typename T>
65 bool parallel(const TLine<T>& l1, const TLine<T>& l2){
66     return abs(vmul(TPoint<T>(l1.a, l1.b), TPoint<T>(l2.a,
        ↪ l2.b))) <= TPoint<T>::eps;
67 }
68 template<typename T>
69 bool equivalent(const TLine<T>& l1, const TLine<T>& l2){
70     return parallel(l1, l2) &&
71         abs(det(l1.b, l1.c, l2.b, l2.c)) <= TPoint<T>::eps &&
72         abs(det(l1.a, l1.c, l2.a, l2.c)) <= TPoint<T>::eps;
73 }

```

• Intersection

```

1 template<typename T>
2 TPoint<T> intersection(const TLine<T>& l1, const TLine<T>&
    ↪ l2){
3     return TPoint<T>(
4         det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b, l2.a,
        ↪ l2.b),

```

```

5         det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b, l2.a,
        ↪ l2.b)
6     );
7 }
8 template<typename T>
9 int sign(const T& x){
10     if (abs(x) <= TPoint<T>::eps) return 0;
11     return x > 0? +1 : -1;
12 }

```

• Area

```

1 template<typename T>
2 T area(const vector<TPoint<T>>& pts){
3     int n = sz(pts);
4     T ans = 0;
5     for (int i = 0; i < n; i++){
6         ans += vmul(pts[i], pts[(i + 1) % n]);
7     }
8     return abs(ans) / 2;
9 }
10 template<typename T>
11 T dist_pp(const TPoint<T>& a, const TPoint<T>& b){
12     return sqrt(sq(a.x - b.x) + sq(a.y - b.y));
13 }
14 template<typename T>
15 TLine<T> perp_line(const TLine<T>& l, const TPoint<T>& p){
16     T na = -l.b, nb = l.a, nc = -na * p.x - nb * p.y;
17     return TLine<T>(na, nb, nc);
18 }

```

• Projection

```

1 template<typename T>
2 TPoint<T> projection(const TPoint<T>& p, const TLine<T>& l){
3     return intersection(l, perp_line(l, p));
4 }
5 template<typename T>
6 T dist_pl(const TPoint<T>& p, const TLine<T>& l){
7     return dist_pp(p, projection(p, l));
8 }
9 template<typename T>
10 struct TRay{
11     TLine<T> l;
12     TPoint<T> start, dirvec;
13     TRay() : l(), start(), dirvec() {}
14     TRay(const TPoint<T>& p1, const TPoint<T>& p2){
15         l = TLine<T>(p1, p2);
16         start = p1, dirvec = p2 - p1;
17     }
18 };
19 template<typename T>
20 bool is_on_line(const TPoint<T>& p, const TLine<T>& l){
21     return abs(l.a * p.x + l.b * p.y + l.c) <= TPoint<T>::eps;
22 }
23 template<typename T>
24 bool is_on_ray(const TPoint<T>& p, const TRay<T>& r){
25     if (is_on_line(p, r.l)){
26         return sign(smul(r.dirvec, TPoint<T>(p - r.start))) != -1;
27     }
28     else return false;
29 }
30 template<typename T>
31 bool is_on_seg(const TPoint<T>& P, const TPoint<T>& A, const
    ↪ TPoint<T>& B){
32     return is_on_ray(P, TRay<T>(A, B)) && is_on_ray(P,
        ↪ TRay<T>(B, A));
33 }
34 template<typename T>
35 T dist_pr(const TPoint<T>& P, const TRay<T>& R){
36     auto H = projection(P, R.l);
37     return is_on_ray(H, R)? dist_pp(P, H) : dist_pp(P, R.start);
38 }
39 template<typename T>
40 T dist_ps(const TPoint<T>& P, const TPoint<T>& A, const
    ↪ TPoint<T>& B){
41     auto H = projection(P, TLine<T>(A, B));
42     if (is_on_seg(H, A, B)) return dist_pp(P, H);

```

```

43     else return min(dist_pp(P, A), dist_pp(P, B));
44 }

    • acw

1 template<typename T>
2 bool acw(const TPoint<T>& A, const TPoint<T>& B){
3     T mul = vmul(A, B);
4     return mul > 0 || abs(mul) <= TPoint<T>::eps;
5 }

    • CW

1 template<typename T>
2 bool cw(const TPoint<T>& A, const TPoint<T>& B){
3     T mul = vmul(A, B);
4     return mul < 0 || abs(mul) <= TPoint<T>::eps;
5 }

    • Convex Hull

1 template<typename T>
2 vector<TPoint<T>> convex_hull(vector<TPoint<T>> pts){
3     sort(all(pts));
4     pts.erase(unique(all(pts)), pts.end());
5     vector<TPoint<T>> up, down;
6     for (auto p : pts){
7         while (sz(up) > 1 && acw(up.end()[-1] - up.end()[-2], p -
↪ up.end()[-2])) up.pop_back();
8         while (sz(down) > 1 && cw(down.end()[-1] - down.end()[-2],
↪ p - down.end()[-2])) down.pop_back();
9         up.pb(p), down.pb(p);
10    }
11    for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
12    return down;
13 }

    • in_triangle

1 template<typename T>
2 bool in_triangle(TPoint<T>& P, TPoint<T>& A, TPoint<T>& B,
↪ TPoint<T>& C){
3     if (is_on_seg(P, A, B) || is_on_seg(P, B, C) || is_on_seg(P,
↪ C, A)) return true;
4     return cw(P - A, B - A) == cw(P - B, C - B) &&
5     cw(P - A, B - A) == cw(P - C, A - C);
6 }

    • prep_convex_poly

1 template<typename T>
2 void prep_convex_poly(vector<TPoint<T>>& pts){
3     rotate(pts.begin(), min_element(all(pts)), pts.end());
4 }

    • in_convex_poly:

```

```

1 // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
2 template<typename T>
3 int in_convex_poly(TPoint<T>& p, vector<TPoint<T>>& pts){
4     int n = sz(pts);
5     if (!n) return 0;
6     if (n <= 2) return is_on_seg(p, pts[0], pts.back());
7     int l = 1, r = n - 1;
8     while (r - l > 1){
9         int mid = (l + r) / 2;
10        if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
11        else r = mid;
12    }
13    if (!in_triangle(p, pts[0], pts[l], pts[l + 1])) return 0;
14    if (is_on_seg(p, pts[l], pts[l + 1]) ||
15        is_on_seg(p, pts[0], pts.back()) ||
16        is_on_seg(p, pts[0], pts[l]))
17    ) return 2;
18    return 1;
19 }

```

• in_simple_poly

```

1 // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
2 template<typename T>
3 int in_simple_poly(TPoint<T> p, vector<TPoint<T>>& pts){
4     int n = sz(pts);
5     bool res = 0;
6     for (int i = 0; i < n; i++){
7         auto a = pts[i], b = pts[(i + 1) % n];
8         if (is_on_seg(p, a, b)) return 2;
9         if (((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p) >
↪ TPoint<T>::eps){
10             res ^= 1;
11         }
12     }
13     return res;
14 }

```

• minkowski_rotate

```

1 template<typename T>
2 void minkowski_rotate(vector<TPoint<T>>& P){
3     int pos = 0;
4     for (int i = 1; i < sz(P); i++){
5         if (abs(P[i].y - P[pos].y) <= TPoint<T>::eps){
6             if (P[i].x < P[pos].x) pos = i;
7         }
8         else if (P[i].y < P[pos].y) pos = i;
9     }
10    rotate(P.begin(), P.begin() + pos, P.end());
11 }

```

• minkowski_sum

```

1 // P and Q are strictly convex, points given in
↪ counterclockwise order
2 template<typename T>
3 vector<TPoint<T>> minkowski_sum(vector<TPoint<T>> P,
↪ vector<TPoint<T>> Q){
4     minkowski_rotate(P);
5     minkowski_rotate(Q);
6     P.pb(P[0]);
7     Q.pb(Q[0]);
8     vector<TPoint<T>> ans;
9     int i = 0, j = 0;
10    while (i < sz(P) - 1 || j < sz(Q) - 1){
11        ans.pb(P[i] + Q[j]);
12        T curmul;
13        if (i == sz(P) - 1) curmul = -1;
14        else if (j == sz(Q) - 1) curmul = +1;
15        else curmul = vmul(P[i + 1] - P[i], Q[j + 1] - Q[j]);
16        if (abs(curmul) < TPoint<T>::eps || curmul > 0) i++;
17        if (abs(curmul) < TPoint<T>::eps || curmul < 0) j++;
18    }
19    return ans;
20 }
21 using Point = TPoint<ll>; using Line = TLine<ll>; using Ray =
↪ TRay<ll>; const ld PI = acos(-1);

```

Strings

```

1 vector<int> prefix_function(string s){
2     int n = sz(s);
3     vector<int> pi(n);
4     for (int i = 1; i < n; i++){
5         int k = pi[i - 1];
6         while (k > 0 && s[i] != s[k]){
7             k = pi[k - 1];
8         }
9         pi[i] = k + (s[i] == s[k]);
10    }
11    return pi;
12 }
13 vector<int> kmp(string s, string k){
14     string st = k + "#" + s;
15     vector<int> res;
16     auto pi = pf(st);
17     for (int i = 0; i < sz(st); i++){
18         if (pi[i] == sz(k)){

```

```

19     res.pb(i - 2 * sz(k));
20 }
21 }
22 return res;
23 }
24 vector<int> z_function(string s){
25     int n = sz(s);
26     vector<int> z(n);
27     int l = 0, r = 0;
28     for (int i = 1; i < n; i++){
29         if (r >= i) z[i] = min(z[i - l], r - i + 1);
30         while (i + z[i] < n && s[z[i]] == s[i + z[i]]){
31             z[i]++;
32         }
33         if (i + z[i] - 1 > r){
34             l = i, r = i + z[i] - 1;
35         }
36     }
37     return z;
38 }

```

Manacher's algorithm

```

1 string longest_palindrome(string& s) {
2     // init "abc" -> "~$a#b#c$"
3     vector<char> t{'~', '#'};
4     for (char c : s) t.push_back(c), t.push_back('#');
5     t.push_back('$');
6     // manacher
7     int n = t.size(), r = 0, c = 0;
8     vector<int> p(n, 0);
9     for (int i = 1; i < n - 1; i++) {
10         if (i < r + c) p[i] = min(p[2 * c - i], r + c - i);
11         while (t[i + p[i] + 1] == t[i - p[i] - 1]) p[i]++;
12         if (i + p[i] > r + c) r = p[i], c = i;
13     }
14     // s[i] -> p[2 * i + 2] (even), p[2 * i + 2] (odd)
15     // output answer
16     int index = 0;
17     for (int i = 0; i < n; i++)
18         if (p[index] < p[i]) index = i;
19     return s.substr((index - p[index]) / 2, p[index]);
20 }

```

Flows

$O(N^2M)$, on unit networks $O(N^{1/2}M)$

```

1 struct FlowEdge {
2     int v, u;
3     ll cap, flow = 0;
4     FlowEdge(int v, int u, ll cap) : v(v), u(u), cap(cap) {}
5 };
6 struct Dinic {
7     const ll flow_inf = 1e18;
8     vector<FlowEdge> edges;
9     vector<vector<int>> adj;
10    int n, m = 0;
11    int s, t;
12    vector<int> level, ptr;
13    queue<int> q;
14    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
15        adj.resize(n);
16        level.resize(n);
17        ptr.resize(n);
18    }
19    void add_edge(int v, int u, ll cap) {
20        edges.emplace_back(v, u, cap);
21        edges.emplace_back(u, v, 0);
22        adj[v].push_back(m);
23        adj[u].push_back(m + 1);
24        m += 2;
25    }
26    bool bfs() {
27        while (!q.empty()) {

```

```

28         int v = q.front();
29         q.pop();
30         for (int id : adj[v]) {
31             if (edges[id].cap - edges[id].flow < 1)
32                 continue;
33             if (level[edges[id].u] != -1)
34                 continue;
35             level[edges[id].u] = level[v] + 1;
36             q.push(edges[id].u);
37         }
38     }
39     return level[t] != -1;
40 }
41 ll dfs(int v, ll pushed) {
42     if (pushed == 0)
43         return 0;
44     if (v == t)
45         return pushed;
46     for (int& cid = ptr[v]; cid < (int)adj[v].size();
47         ↪ cid++) {
48         int id = adj[v][cid];
49         int u = edges[id].u;
50         if (level[v] + 1 != level[u] || edges[id].cap -
51         ↪ edges[id].flow < 1)
52             continue;
53         ll tr = dfs(u, min(pushed, edges[id].cap -
54         ↪ edges[id].flow));
55         if (tr == 0)
56             continue;
57         edges[id].flow += tr;
58         edges[id ^ 1].flow -= tr;
59         return tr;
60     }
61     return 0;
62 }
63 ll flow() {
64     ll f = 0;
65     while (true) {
66         fill(level.begin(), level.end(), -1);
67         level[s] = 0;
68         q.push(s);
69         if (!bfs())
70             break;
71         fill(ptr.begin(), ptr.end(), 0);
72         while (ll pushed = dfs(s, flow_inf)) {
73             f += pushed;
74         }
75     }
76     return f;
77 }
78 // To recover flow through original edges: iterate over even
79 ↪ indices in edges.

```

MCMF – maximize flow, then minimize its cost. $O(Fmn)$.

```

1 #include <ext/pb_ds/priority_queue.hpp>
2 template <typename T, typename C>
3 class MCMF {
4     public:
5         static constexpr T eps = (T) 1e-9;
6
7         struct edge {
8             int from;
9             int to;
10            T c;
11            T f;
12            C cost;
13        };
14
15        int n;
16        vector<vector<int>> g;
17        vector<edge> edges;
18        vector<C> d;
19        vector<C> pot;

```

```

20 __gnu_pbds::priority_queue<pair<C, int>> q;
21 vector<typename decltype(q)::point_iterator> its;
22 vector<int> pe;
23 const C INF_C = numeric_limits<C>::max() / 2;
24
25 explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0),
↳ its(n), pe(n) {}
26
27 int add(int from, int to, T forward_cap, C edge_cost, T
↳ backward_cap = 0) {
28     assert(0 <= from && from < n && 0 <= to && to < n);
29     assert(forward_cap >= 0 && backward_cap >= 0);
30     int id = static_cast<int>(edges.size());
31     g[from].push_back(id);
32     edges.push_back({from, to, forward_cap, 0, edge_cost});
33     g[to].push_back(id + 1);
34     edges.push_back({to, from, backward_cap, 0, -edge_cost});
35     return id;
36 }
37
38 void expath(int st) {
39     fill(d.begin(), d.end(), INF_C);
40     q.clear();
41     fill(its.begin(), its.end(), q.end());
42     its[st] = q.push({pot[st], st});
43     d[st] = 0;
44     while (!q.empty()) {
45         int i = q.top().second;
46         q.pop();
47         its[i] = q.end();
48         for (int id : g[i]) {
49             const edge &e = edges[id];
50             int j = e.to;
51             if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
52                 d[j] = d[i] + e.cost;
53                 pe[j] = id;
54                 if (its[j] == q.end()) {
55                     its[j] = q.push({pot[j] - d[j], j});
56                 } else {
57                     q.modify(its[j], {pot[j] - d[j], j});
58                 }
59             }
60         }
61     }
62     swap(d, pot);
63 }
64
65 pair<T, C> max_flow(int st, int fin) {
66     T flow = 0;
67     C cost = 0;
68     bool ok = true;
69     for (auto& e : edges) {
70         if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to]
↳ < 0) {
71             ok = false;
72             break;
73         }
74     }
75     if (ok) {
76         expath(st);
77     } else {
78         vector<int> deg(n, 0);
79         for (int i = 0; i < n; i++) {
80             for (int eid : g[i]) {
81                 auto& e = edges[eid];
82                 if (e.c - e.f > eps) {
83                     deg[e.to] += 1;
84                 }
85             }
86         }
87         vector<int> que;
88         for (int i = 0; i < n; i++) {
89             if (deg[i] == 0) {
90                 que.push_back(i);
91             }
92         }
93         for (int b = 0; b < (int) que.size(); b++) {
94             for (int eid : g[que[b]]) {
95                 auto& e = edges[eid];
96                 if (e.c - e.f > eps) {
97                     deg[e.to] -= 1;
98                     if (deg[e.to] == 0) {
99                         que.push_back(e.to);
100                     }
101                 }
102             }
103         }
104         fill(pot.begin(), pot.end(), INF_C);
105         pot[st] = 0;
106         if (static_cast<int>(que.size()) == n) {
107             for (int v : que) {
108                 if (pot[v] < INF_C) {
109                     for (int eid : g[v]) {
110                         auto& e = edges[eid];
111                         if (e.c - e.f > eps) {
112                             if (pot[v] + e.cost < pot[e.to]) {
113                                 pot[e.to] = pot[v] + e.cost;
114                                 pe[e.to] = eid;
115                             }
116                         }
117                     }
118                 }
119             }
120         } else {
121             que.assign(1, st);
122             vector<bool> in_queue(n, false);
123             in_queue[st] = true;
124             for (int b = 0; b < (int) que.size(); b++) {
125                 int i = que[b];
126                 in_queue[i] = false;
127                 for (int id : g[i]) {
128                     const edge &e = edges[id];
129                     if (e.c - e.f > eps && pot[i] + e.cost <
↳ pot[e.to]) {
130                         pot[e.to] = pot[i] + e.cost;
131                         pe[e.to] = id;
132                         if (!in_queue[e.to]) {
133                             que.push_back(e.to);
134                             in_queue[e.to] = true;
135                         }
136                     }
137                 }
138             }
139         }
140     }
141     while (pot[fin] < INF_C) {
142         T push = numeric_limits<T>::max();
143         int v = fin;
144         while (v != st) {
145             const edge &e = edges[pe[v]];
146             push = min(push, e.c - e.f);
147             v = e.from;
148         }
149         v = fin;
150         while (v != st) {
151             edge &e = edges[pe[v]];
152             e.f += push;
153             edge &back = edges[pe[v] ^ 1];
154             back.f -= push;
155             v = e.from;
156         }
157         flow += push;
158         cost += push * pot[fin];
159         expath(st);
160     }
161     return {flow, cost};
162 }
163 };
164
165 // Examples: MCMF<int, int> g(n); g.add(u,v,c,w,0);
166 ↳ g.max_flow(s,t).
167 // To recover flow through original edges: iterate over even
168 ↳ indices in edges.

```

Graphs

Kuhn's algorithm for bipartite matching

```
1  /*
2  The graph is split into 2 halves of n1 and n2 vertices.
3  Complexity: O(n1 * m). Usually runs much faster. MUCH
4  ↪ FASTER!!!
5  */
6  const int N = 305;
7  vector<int> g[N]; // Stores edges from left half to right.
8  bool used[N]; // Stores if vertex from left half is used.
9  int mt[N]; // For every vertex in right half, stores to which
10 ↪ vertex in left half it's matched (-1 if not matched).
11
12 bool try_dfs(int v){
13     if (used[v]) return false;
14     used[v] = 1;
15     for (auto u : g[v]){
16         if (mt[u] == -1 || try_dfs(mt[u])){
17             mt[u] = v;
18             return true;
19         }
20     }
21     return false;
22 }
23
24 int main(){
25     // .....
26     for (int i = 1; i <= n2; i++) mt[i] = -1;
27     for (int i = 1; i <= n1; i++) used[i] = 0;
28     for (int i = 1; i <= n1; i++){
29         if (try_dfs(i)){
30             for (int j = 1; j <= n1; j++) used[j] = 0;
31         }
32     }
33     vector<pair<int, int>> ans;
34     for (int i = 1; i <= n2; i++){
35         if (mt[i] != -1) ans.pb({mt[i], i});
36     }
37
38     // Finding maximal independent set: size = # of nodes - # of
39     ↪ edges in matching.
40     // To construct: launch Kuhn-like DFS from unmatched nodes in
41     ↪ the left half.
42     // Independent set = visited nodes in left half + unvisited in
43     ↪ right half.
44     // Finding minimal vertex cover: complement of maximal
45     ↪ independent set.
```

Hungarian algorithm for Assignment Problem

- Given a 1-indexed $(n \times m)$ matrix A , select a number in each row such that each column has at most 1 number selected, and the sum of the selected numbers is minimized.

```
1  int INF = 1e9; // constant greater than any number in the
2  ↪ matrix
3  vector<int> u(n+1), v(m+1), p(m+1), way(m+1);
4  for (int i=1; i<=n; ++i) {
5      p[0] = i;
6      int j0 = 0;
7      vector<int> minv (m+1, INF);
8      vector<bool> used (m+1, false);
9      do {
10         used[j0] = true;
11         int i0 = p[j0], delta = INF, j1;
12         for (int j=1; j<=m; ++j)
13             if (!used[j]) {
14                 int cur = A[i0][j]-u[i0]-v[j];
15                 if (cur < minv[j])
```

```
16                     minv[j] = cur, way[j] = j0;
17                     if (minv[j] < delta)
18                         delta = minv[j], j1 = j;
19                 }
20             } while (p[j0] != 0);
21             do {
22                 int j1 = way[j0];
23                 p[j0] = p[j1];
24                 j0 = j1;
25             } while (j0);
26         } while (j0);
27     }
28     vector<int> ans (n+1); // ans[i] stores the column selected
29     ↪ for row i
30     for (int j=1; j<=m; ++j)
31         ans[p[j]] = j;
32     int cost = -v[0]; // the total cost of the matching
```

Dijkstra's Algorithm

```
1  priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
2  ↪ greater<pair<ll, ll>>> q;
3  dist[start] = 0;
4  q.push({0, start});
5  while (!q.empty()){
6      auto [d, v] = q.top();
7      q.pop();
8      if (d != dist[v]) continue;
9      for (auto [u, w] : g[v]){
10         if (dist[u] > dist[v] + w){
11             dist[u] = dist[v] + w;
12             q.push({dist[u], u});
13         }
14     }
```

Eulerian Cycle DFS

```
1  void dfs(int v){
2      while (!g[v].empty()){
3          int u = g[v].back();
4          g[v].pop_back();
5          dfs(u);
6          ans.pb(v);
7      }
8  }
```

SCC and 2-SAT

```
1  void scc(vector<vector<int>>& g, int* idx) {
2      int n = g.size(), ct = 0;
3      int out[n];
4      vector<int> ginv[n];
5      memset(out, -1, sizeof out);
6      memset(idx, -1, n * sizeof(int));
7      function<void(int)> dfs = [&](int cur) {
8          out[cur] = INT_MAX;
9          for(int v : g[cur]) {
10             ginv[v].push_back(cur);
11             if(out[v] == -1) dfs(v);
12         }
13         ct++; out[cur] = ct;
14     };
15     vector<int> order;
16     for(int i = 0; i < n; i++) {
17         order.push_back(i);
18         if(out[i] == -1) dfs(i);
19     }
20     sort(order.begin(), order.end(), [&](int& u, int& v) {
21         return out[u] > out[v];
22     });
```



```

22     });
23     ct = 0;
24     stack<int> s;
25     auto dfs2 = [&](int start) {
26         s.push(start);
27         while(!s.empty()) {
28             int cur = s.top();
29             s.pop();
30             idx[cur] = ct;
31             for(int v : ginv[cur])
32                 if(idx[v] == -1) s.push(v);
33         }
34     };
35     for(int v : order) {
36         if(idx[v] == -1) {
37             dfs2(v);
38             ct++;
39         }
40     }
41 }
42
43 // 0 => impossible, 1 => possible
44 pair<int, vector<int>> sat2(int n, vector<pair<int, int>>&
45     ↪ clauses) {
46     vector<int> ans(n);
47     vector<vector<int>> g(2*n + 1);
48     for(auto [x, y] : clauses) {
49         x = x < 0 ? -x + n : x;
50         y = y < 0 ? -y + n : y;
51         int nx = x <= n ? x + n : x - n;
52         int ny = y <= n ? y + n : y - n;
53         g[nx].push_back(y);
54         g[ny].push_back(x);
55     }
56     int idx[2*n + 1];
57     scc(g, idx);
58     for(int i = 1; i <= n; i++) {
59         if(idx[i] == idx[i + n]) return {0, {}};
60         ans[i - 1] = idx[i + n] < idx[i];
61     }
62     return {1, ans};
63 }

```

Finding Bridges

```

1  /*
2  Bridges.
3  Results are stored in a map "is_bridge".
4  For each connected component, call "dfs(starting vertex,
5  ↪ starting vertex)".
6  */
7  const int N = 2e5 + 10; // Careful with the constant!
8
9  vector<int> g[N];
10 int tin[N], fup[N], timer;
11 map<pair<int, int>, bool> is_bridge;
12
13 void dfs(int v, int p){
14     tin[v] = ++timer;
15     fup[v] = tin[v];
16     for (auto u : g[v]){
17         if (!tin[u]){
18             dfs(u, v);
19             if (fup[u] > tin[v]){
20                 is_bridge[{u, v}] = is_bridge[{v, u}] = true;
21             }
22             fup[v] = min(fup[v], fup[u]);
23         }
24         else{
25             if (u != p) fup[v] = min(fup[v], tin[u]);
26         }
27     }
28 }

```

Virtual Tree

```

1  // order stores the nodes in the queried set
2  sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
3  int m = sz(order);
4  for (int i = 1; i < m; i++){
5       order.pb(lca(order[i], order[i - 1]));
6  }
7  sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
8  order.erase(unique(all(order)), order.end());
9  vector<int> stk[order[0]];
10 for (int i = 1; i < sz(order); i++){
11     int v = order[i];
12     while (tout[stk.back()] < tout[v]) stk.pop_back();
13     int u = stk.back();
14     vg[u].pb({v, dep[v] - dep[u]});
15     stk.pb(v);
16 }

```

HLD on Edges DFS

```

1  void dfs1(int v, int p, int d){
2      par[v] = p;
3      for (auto e : g[v]){
4          if (e.fi == p){
5              g[v].erase(find(all(g[v]), e));
6              break;
7          }
8      }
9      dep[v] = d;
10     sz[v] = 1;
11     for (auto [u, c] : g[v]){
12         dfs1(u, v, d + 1);
13         sz[v] += sz[u];
14     }
15     if (!g[v].empty()) iter_swap(g[v].begin(),
16     ↪ max_element(all(g[v]), comp));
17 }
18 void dfs2(int v, int rt, int c){
19     pos[v] = sz(a);
20     a.pb(c);
21     root[v] = rt;
22     for (int i = 0; i < sz(g[v]); i++){
23         auto [u, c] = g[v][i];
24         if (!i) dfs2(u, rt, c);
25         else dfs2(u, u, c);
26     }
27 }
28 int getans(int u, int v){
29     int res = 0;
30     for (; root[u] != root[v]; v = par[root[v]]){
31         if (dep[root[u]] > dep[root[v]]) swap(u, v);
32         res = max(res, rmq(0, 0, n - 1, pos[root[v]], pos[v]));
33     }
34     if (pos[u] > pos[v]) swap(u, v);
35     return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));
36 }

```

Centroid Decomposition

```

1  vector<char> res(n), seen(n), sz(n);
2  function<int(int, int)> get_size = [&](int node, int fa) {
3      sz[node] = 1;
4      for (auto& ne : g[node]) {
5          if (ne == fa || seen[ne]) continue;
6          sz[node] += get_size(ne, node);
7      }
8      return sz[node];
9  };
10 function<int(int, int, int)> find_centroid = [&](int node, int
11     ↪ fa, int t) {
12     for (auto& ne : g[node])
13         if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
14     ↪ find_centroid(ne, node, t);
15     return node;
16 };

```



```

15 function<void(int, char)> solve = [&](int node, char cur) {
16     get_size(node, -1); auto c = find_centroid(node, -1,
    ↪ sz[node]);
17     seen[c] = 1, res[c] = cur;
18     for (auto& ne : g[c]) {
19         if (seen[ne]) continue;
20         solve(ne, char(cur + 1)); // we can pass c here to build
    ↪ tree
21     }
22 };

```

Math

Binary exponentiation

```

1 ll power(ll a, ll b){
2     ll res = 1;
3     for (; b; a = a * a % MOD, b >>= 1){
4         if (b & 1) res = res * a % MOD;
5     }
6     return res;
7 }

```

Matrix Exponentiation: $O(n^3 \log b)$

```

1 const int N = 100, MOD = 1e9 + 7;
2
3 struct matrix{
4     ll m[N][N];
5     int n;
6     matrix(){
7         n = N;
8         memset(m, 0, sizeof(m));
9     };
10    matrix(int n_){
11        n = n_;
12        memset(m, 0, sizeof(m));
13    };
14    matrix(int n_, ll val){
15        n = n_;
16        memset(m, 0, sizeof(m));
17        for (int i = 0; i < n; i++){
18            m[i][i] = val;
19        }
20
21        matrix operator* (matrix oth){
22            matrix res(n);
23            for (int i = 0; i < n; i++){
24                for (int j = 0; j < n; j++){
25                    res.m[i][j] = (res.m[i][j] + m[i][k] * oth.m[k][j])
    ↪ % MOD;
26                }
27            }
28            return res;
29        }
30    };
31 };
32
33 matrix power(matrix a, ll b){
34     matrix res(a.n, 1);
35     for (; b; a = a * a, b >>= 1){
36         if (b & 1) res = res * a;
37     }
38     return res;
39 }

```

Extended Euclidean Algorithm

```

1 // gives (x, y) for ax + by = g
2 // solutions given (x0, y0): a(x0 + kb/g) + b(y0 - ka/g) = g
3 int gcd(int a, int b, int& x, int& y) {
4     x = 1, y = 0; int sum1 = a;
5     int x2 = 0, y2 = 1, sum2 = b;
6     while (sum2) {

```

```

7         int q = sum1 / sum2;
8         tie(x, x2) = make_tuple(x2, x - q * x2);
9         tie(y, y2) = make_tuple(y2, y - q * y2);
10        tie(sum1, sum2) = make_tuple(sum2, sum1 - q * sum2);
11    }
12    return sum1;
13 }

```

Linear Sieve

• Mobius Function

```

1 vector<int> prime;
2 bool is_composite[MAX_N];
3 int mu[MAX_N];
4
5 void sieve(int n){
6     fill(is_composite, is_composite + n, 0);
7     mu[1] = 1;
8     for (int i = 2; i < n; i++){
9         if (!is_composite[i]){
10            prime.push_back(i);
11            mu[i] = -1; //i is prime
12        }
13        for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
14            is_composite[i * prime[j]] = true;
15            if (i % prime[j] == 0){
16                mu[i * prime[j]] = 0; //prime[j] divides i
17                break;
18            } else {
19                mu[i * prime[j]] = -mu[i]; //prime[j] does not divide i
20            }
21        }
22    }
23 }

```

• Euler's Totient Function

```

1 vector<int> prime;
2 bool is_composite[MAX_N];
3 int phi[MAX_N];
4
5 void sieve(int n){
6     fill(is_composite, is_composite + n, 0);
7     phi[1] = 1;
8     for (int i = 2; i < n; i++){
9         if (!is_composite[i]){
10            prime.push_back(i);
11            phi[i] = i - 1; //i is prime
12        }
13        for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
14            is_composite[i * prime[j]] = true;
15            if (i % prime[j] == 0){
16                phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
    ↪ divides i
17                break;
18            } else {
19                phi[i * prime[j]] = phi[i] * phi[prime[j]]; //prime[j]
    ↪ does not divide i
20            }
21        }
22    }
23 }

```

Gaussian Elimination

```

1 bool is_0(Z v) { return v.x == 0; }
2 Z abs(Z v) { return v; }
3 bool is_0(double v) { return abs(v) < 1e-9; }
4
5 // 1 => unique solution, 0 => no solution, -1 => multiple
    ↪ solutions
6 template <typename T>
7 int gaussian_elimination(vector<vector<T>> &a, int limit) {
8     if (a.empty() || a[0].empty()) return -1;
9     int h = (int)a.size(), w = (int)a[0].size(), r = 0;
10    for (int c = 0; c < limit; c++) {

```

```

11     int id = -1;
12     for (int i = r; i < h; i++) {
13         if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
↪ abs(a[i][c]))) {
14             id = i;
15         }
16     }
17     if (id == -1) continue;
18     if (id > r) {
19         swap(a[r], a[id]);
20         for (int j = c; j < w; j++) a[id][j] = -a[id][j];
21     }
22     vector<int> nonzero;
23     for (int j = c; j < w; j++) {
24         if (!is_0(a[r][j])) nonzero.push_back(j);
25     }
26     T inv_a = 1 / a[r][c];
27     for (int i = r + 1; i < h; i++) {
28         if (is_0(a[i][c])) continue;
29         T coeff = -a[i][c] * inv_a;
30         for (int j : nonzero) a[i][j] += coeff * a[r][j];
31     }
32     ++r;
33 }
34 for (int row = h - 1; row >= 0; row--) {
35     for (int c = 0; c < limit; c++) {
36         if (!is_0(a[row][c])) {
37             T inv_a = 1 / a[row][c];
38             for (int i = row - 1; i >= 0; i--) {
39                 if (is_0(a[i][c])) continue;
40                 T coeff = -a[i][c] * inv_a;
41                 for (int j = c; j < w; j++) a[i][j] += coeff *
↪ a[row][j];
42             }
43             break;
44         }
45     }
46     // not-free variables: only it on its line
47     for (int i = r; i < h; i++) if (!is_0(a[i][limit])) return 0;
48     return (r == limit) ? 1 : -1;
49 }
50
51 template <typename T>
52 pair<int, vector<T>> solve_linear(vector<vector<T>> a, const
↪ vector<T> &b, int w) {
53     int h = (int)a.size();
54     for (int i = 0; i < h; i++) a[i].push_back(b[i]);
55     int sol = gaussian_elimination(a, w);
56     if (!sol) return {0, vector<T>()};
57     vector<T> x(w, 0);
58     for (int i = 0; i < h; i++) {
59         for (int j = 0; j < w; j++) {
60             if (!is_0(a[i][j])) {
61                 x[j] = a[i][w] / a[i][j];
62                 break;
63             }
64         }
65     }
66     return {sol, x};
67 }

```

NTT

```

1 void ntt(vector<ll>& a, int f) {
2     int n = (int)a.size();
3     vector<ll> w(n);
4     vector<int> rev(n);
5     for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2) | ((i
↪ & 1) * (n / 2));
6     for (int i = 0; i < n; i++) {
7         if (i < rev[i]) swap(a[i], a[rev[i]]);
8     }
9     ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
10    w[0] = 1;
11    for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn % MOD;
12    for (int mid = 1; mid < n; mid *= 2) {
13        for (int i = 0; i < n; i += 2 * mid) {

```

```

14            for (int j = 0; j < mid; j++) {
15                ll x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid)
↪ * j] % MOD;
16                a[i + j] = (x + y) % MOD, a[i + j + mid] = (x + MOD -
↪ y) % MOD;
17            }
18        }
19    }
20    if (f) {
21        ll iv = power(n, MOD - 2);
22        for (auto& x : a) x = x * iv % MOD;
23    }
24 }
25 vector<ll> mul(vector<ll> a, vector<ll> b) {
26     int n = 1, m = (int)a.size() + (int)b.size() - 1;
27     while (n < m) n *= 2;
28     a.resize(n), b.resize(n);
29     ntt(a, 0), ntt(b, 0); // if squaring, you can save one NTT
↪ here
30     for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
31     ntt(a, 1);
32     a.resize(m);
33     return a;
34 }

```

FFT

```

1 const ld PI = acos(-1);
2 auto mul = [&](const vector<ld>& aa, const vector<ld>& bb) {
3     int n = (int)aa.size(), m = (int)bb.size(), bit = 1;
4     while ((1 << bit) < n + m - 1) bit++;
5     int len = 1 << bit;
6     vector<complex<ld>> a(len), b(len);
7     vector<int> rev(len);
8     for (int i = 0; i < n; i++) a[i].real(aa[i]);
9     for (int i = 0; i < m; i++) b[i].real(bb[i]);
10    for (int i = 0; i < len; i++) rev[i] = (rev[i >> 1] >> 1) |
↪ ((i & 1) << (bit - 1));
11    auto fft = [&](vector<complex<ld>>& p, int inv) {
12        for (int i = 0; i < len; i++)
13            if (i < rev[i]) swap(p[i], p[rev[i]]);
14        for (int mid = 1; mid < len; mid *= 2) {
15            auto w1 = complex<ld>(cos(PI / mid), (inv ? -1 : 1) *
↪ sin(PI / mid));
16            for (int i = 0; i < len; i += mid * 2) {
17                auto wk = complex<ld>(1, 0);
18                for (int j = 0; j < mid; j++, wk = wk * w1) {
19                    auto x = p[i + j], y = wk * p[i + j + mid];
20                    p[i + j] = x + y, p[i + j + mid] = x - y;
21                }
22            }
23        }
24        if (inv == 1) {
25            for (int i = 0; i < len; i++) p[i].real(p[i].real() /
↪ len);
26        }
27    };
28    fft(a, 0), fft(b, 0);
29    for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
30    fft(a, 1);
31    a.resize(n + m - 1);
32    vector<ld> res(n + m - 1);
33    for (int i = 0; i < n + m - 1; i++) res[i] = a[i].real();
34    return res;
35 };

```

is_prime

- (Miller-Rabin primality test)

```

1 typedef __int128_t i128;
2
3 i128 power(i128 a, i128 b, i128 MOD = 1, i128 res = 1) {
4     for (; b; b /= 2, (a *= a) %= MOD)
5         if (b & 1) (res *= a) %= MOD;
6     return res;

```

```

7 }
8
9 bool is_prime(ll n) {
10     if (n < 2) return false;
11     static constexpr int A[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
12     int s = __builtin_ctzll(n - 1);
13     ll d = (n - 1) >> s;
14     for (auto a : A) {
15         if (a == n) return true;
16         ll x = (ll)power(a, d, n);
17         if (x == 1 || x == n - 1) continue;
18         bool ok = false;
19         for (int i = 0; i < s - 1; ++i) {
20             x = ll((i128)x * x % n); // potential overflow!
21             if (x == n - 1) {
22                 ok = true;
23                 break;
24             }
25         }
26         if (!ok) return false;
27     }
28     return true;
29 }
30
31 typedef __int128_t i128;
32
33 ll pollard_rho(ll x) {
34     ll s = 0, t = 0, c = rng() % (x - 1) + 1;
35     ll stp = 0, goal = 1, val = 1;
36     for (goal = 1;; goal *= 2, s = t, val = 1) {
37         for (stp = 1; stp <= goal; ++stp) {
38             t = ll(((i128)t * t + c) % x);
39             val = ll(((i128)val * abs(t - s) % x);
40             if ((stp % 127) == 0) {
41                 ll d = gcd(val, x);
42                 if (d > 1) return d;
43             }
44         }
45         ll d = gcd(val, x);
46         if (d > 1) return d;
47     }
48 }
49
50 ll get_max_factor(ll _x) {
51     ll max_factor = 0;
52     function<void(ll)> fac = [&](ll x) {
53         if (x <= max_factor || x < 2) return;
54         if (is_prime(x)) {
55             max_factor = max_factor > x ? max_factor : x;
56             return;
57         }
58         ll p = x;
59         while (p >= x) p = pollard_rho(x);
60         while ((x % p) == 0) x /= p;
61         fac(x), fac(p);
62     };
63     fac(_x);
64     return max_factor;
65 }

```

Berlekamp-Massey

- Recovers any n -order linear recurrence relation from the first $2n$ terms of the sequence.
- Input s is the sequence to be analyzed.
- Output c is the shortest sequence c_1, \dots, c_n , such that

$$s_m = \sum_{i=1}^n c_i \cdot s_{m-i}, \text{ for all } m \geq n.$$

- Be careful since c is returned in 0-based indexation.
- Complexity: $O(N^2)$

```

1 vector<ll> berlekamp_massey(vector<ll> s) {
2     int n = sz(s), l = 0, m = 1;

```

```

3     vector<ll> b(n), c(n);
4     ll ldd = b[0] = c[0] = 1;
5     for (int i = 0; i < n; i++, m++) {
6         ll d = s[i];
7         for (int j = 1; j <= l; j++) d = (d + c[j] * s[i - j]) %
8         MOD;
9         if (d == 0) continue;
10        vector<ll> temp = c;
11        ll coef = d * power(ldd, MOD - 2) % MOD;
12        for (int j = m; j < n; j++){
13            c[j] = (c[j] + MOD - coef * b[j - m]) % MOD;
14            if (c[j] < 0) c[j] += MOD;
15        }
16        if (2 * l <= i) {
17            l = i + 1 - l;
18            b = temp;
19            ldd = d;
20            m = 0;
21        }
22        c.resize(l + 1);
23        c.erase(c.begin());
24        for (ll &x : c)
25            x = (MOD - x) % MOD;
26        return c;
27    }

```

Calculating k -th term of a linear recurrence

- Given the first n terms s_0, s_1, \dots, s_{n-1} and the sequence c_1, c_2, \dots, c_n such that

$$s_m = \sum_{i=1}^n c_i \cdot s_{m-i}, \text{ for all } m \geq n,$$

the function `calc_kth` computes s_k .

- Complexity: $O(n^2 \log k)$

```

1 vector<ll> poly_mult_mod(vector<ll> p, vector<ll> q,
2     vector<ll> &c) {
3     vector<ll> ans(sz(p) + sz(q) - 1);
4     for (int i = 0; i < sz(p); i++) {
5         for (int j = 0; j < sz(q); j++) {
6             ans[i + j] = (ans[i + j] + p[i] * q[j]) % MOD;
7         }
8     }
9     int n = sz(ans), m = sz(c);
10    for (int i = n - 1; i >= m; i--) {
11        for (int j = 0; j < m; j++) {
12            ans[i - 1 - j] = (ans[i - 1 - j] + c[j] * ans[i]) % MOD;
13        }
14    }
15    ans.resize(m);
16    return ans;
17 }
18
19 ll calc_kth(vector<ll> s, vector<ll> c, ll k) {
20     assert(sz(s) >= sz(c)); // size of s can be greater than c,
21     // but not less
22     if (k < sz(s)) return s[k];
23     vector<ll> res{1};
24     for (vector<ll> poly = {0, 1}; k; poly = poly_mult_mod(poly,
25     poly, c), k >>= 1) {
26         if (k & 1) res = poly_mult_mod(res, poly, c);
27     }
28     ll ans = 0;
29     for (int i = 0; i < min(sz(res), sz(c)); i++) ans = (ans +
30     s[i] * res[i]) % MOD;
31     return ans;
32 }

```

Data Structures

Fenwick Tree

```
1 ll sum(int r) {
2     ll ret = 0;
3     for (; r >= 0; r = (r & r + 1) - 1) ret += bit[r];
4     return ret;
5 }
6 void add(int idx, ll delta) {
7     for (; idx < n; idx |= idx + 1) bit[idx] += delta;
8 }
```

Lazy Propagation SegTree

```
1 // Clear: clear() or build()
2 const int N = 2e5 + 10; // Change the constant!
3 template<typename T>
4 struct LazySegTree{
5     T t[4 * N];
6     T lazy[4 * N];
7     int n;
8
9     // Change these functions, default return, and lazy mark.
10    T default_return = 0, lazy_mark = numeric_limits<T>::min();
11    // Lazy mark is how the algorithm will identify that no
12    // propagation is needed.
13    function<T(T, T)> f = [&] (T a, T b){
14        return a + b;
15    };
16    // f_on_seg calculates the function f, knowing the lazy
17    // value on segment,
18    // segment's size and the previous value.
19    // The default is segment modification for RSQ. For
20    // increments change to:
21    // return cur_seg_val + seg_size * lazy_val;
22    // For RMQ. Modification: return lazy_val; Increments:
23    // return cur_seg_val + lazy_val;
24    function<T(T, int, T)> f_on_seg = [&] (T cur_seg_val, int
25    seg_size, T lazy_val){
26        return seg_size * lazy_val;
27    };
28    // upd_lazy updates the value to be propagated to child
29    // segments.
30    // Default: modification. For increments change to:
31    // lazy[v] = (lazy[v] == lazy_mark? val : lazy[v] +
32    // val);
33    function<void(int, T)> upd_lazy = [&] (int v, T val){
34        lazy[v] = val;
35    };
36    // Tip: for "get element on single index" queries, use max()
37    // on segment: no overflows.
38
39    LazySegTree(int n_) : n(n_) {
40        clear(n);
41    }
42
43    void build(int v, int tl, int tr, vector<T>& a){
44        if (tl == tr) {
45            t[v] = a[tl];
46            return;
47        }
48        int tm = (tl + tr) / 2;
49        // left child: [tl, tm]
50        // right child: [tm + 1, tr]
51        build(2 * v + 1, tl, tm, a);
52        build(2 * v + 2, tm + 1, tr, a);
53        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
54    }
55
56    LazySegTree(vector<T>& a){
57        build(a);
58    }
59
60    void push(int v, int tl, int tr){
61        if (lazy[v] == lazy_mark) return;
62        if (tl == tr) {
63            t[v] = lazy[v];
64            return;
65        }
66        push(v, tl, (tl + tr) / 2);
67        push(v, (tl + tr) / 2 + 1, tr);
68        t[v] = f(t[v], t[v]);
69    }
70
71    T query(int l, int r) {
72        return query(0, 0, n - 1, l, r);
73    }
74
75    void modify(int l, int r, T val) {
76        modify(0, 0, n - 1, l, r, val);
77    }
78
79    T query(int l, int r) {
80        return query(0, 0, n - 1, l, r);
81    }
82
83    T get(int pos) {
84        return query(pos, pos);
85    }
86
87    // Change clear() function to t.clear() if using
88    // unordered_map for SegTree!!!
89    void clear(int n_) {
90        n = n_;
91        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
92        lazy_mark;
93    }
94
95    void build(vector<T>& a) {
96        n = sz(a);
97        clear(n);
98        build(0, 0, n - 1, a);
99    }
100
101    void build(int v, int tl, int tr, vector<T>& a) {
102        if (tl == tr) {
103            t[v] = a[tl];
104            return;
105        }
106        int tm = (tl + tr) / 2;
107        // left child: [tl, tm]
108        // right child: [tm + 1, tr]
109        build(2 * v + 1, tl, tm, a);
110        build(2 * v + 2, tm + 1, tr, a);
111        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
112    }
113
114    LazySegTree(vector<T>& a) {
115        build(a);
116    }
117
118    void push(int v, int tl, int tr) {
119        if (lazy[v] == lazy_mark) return;
120        if (tl == tr) {
121            t[v] = lazy[v];
122            return;
123        }
124        push(v, tl, (tl + tr) / 2);
125        push(v, (tl + tr) / 2 + 1, tr);
126        t[v] = f(t[v], t[v]);
127    }
128
129    T query(int l, int r) {
130        return query(0, 0, n - 1, l, r);
131    }
132
133    void modify(int l, int r, T val) {
134        modify(0, 0, n - 1, l, r, val);
135    }
136
137    T query(int l, int r) {
138        return query(0, 0, n - 1, l, r);
139    }
140
141    T get(int pos) {
142        return query(pos, pos);
143    }
144
145    // Change clear() function to t.clear() if using
146    // unordered_map for SegTree!!!
147    void clear(int n_) {
148        n = n_;
149        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
150        lazy_mark;
151    }
152
153    void build(vector<T>& a) {
154        n = sz(a);
155        clear(n);
156        build(0, 0, n - 1, a);
157    }
158
159    void build(int v, int tl, int tr, vector<T>& a) {
160        if (tl == tr) {
161            t[v] = a[tl];
162            return;
163        }
164        int tm = (tl + tr) / 2;
165        // left child: [tl, tm]
166        // right child: [tm + 1, tr]
167        build(2 * v + 1, tl, tm, a);
168        build(2 * v + 2, tm + 1, tr, a);
169        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
170    }
171
172    LazySegTree(vector<T>& a) {
173        build(a);
174    }
175
176    void push(int v, int tl, int tr) {
177        if (lazy[v] == lazy_mark) return;
178        if (tl == tr) {
179            t[v] = lazy[v];
180            return;
181        }
182        push(v, tl, (tl + tr) / 2);
183        push(v, (tl + tr) / 2 + 1, tr);
184        t[v] = f(t[v], t[v]);
185    }
186
187    T query(int l, int r) {
188        return query(0, 0, n - 1, l, r);
189    }
190
191    void modify(int l, int r, T val) {
192        modify(0, 0, n - 1, l, r, val);
193    }
194
195    T query(int l, int r) {
196        return query(0, 0, n - 1, l, r);
197    }
198
199    T get(int pos) {
200        return query(pos, pos);
201    }
202
203    // Change clear() function to t.clear() if using
204    // unordered_map for SegTree!!!
205    void clear(int n_) {
206        n = n_;
207        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
208        lazy_mark;
209    }
210
211    void build(vector<T>& a) {
212        n = sz(a);
213        clear(n);
214        build(0, 0, n - 1, a);
215    }
216
217    void build(int v, int tl, int tr, vector<T>& a) {
218        if (tl == tr) {
219            t[v] = a[tl];
220            return;
221        }
222        int tm = (tl + tr) / 2;
223        // left child: [tl, tm]
224        // right child: [tm + 1, tr]
225        build(2 * v + 1, tl, tm, a);
226        build(2 * v + 2, tm + 1, tr, a);
227        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
228    }
229
230    LazySegTree(vector<T>& a) {
231        build(a);
232    }
233
234    void push(int v, int tl, int tr) {
235        if (lazy[v] == lazy_mark) return;
236        if (tl == tr) {
237            t[v] = lazy[v];
238            return;
239        }
240        push(v, tl, (tl + tr) / 2);
241        push(v, (tl + tr) / 2 + 1, tr);
242        t[v] = f(t[v], t[v]);
243    }
244
245    T query(int l, int r) {
246        return query(0, 0, n - 1, l, r);
247    }
248
249    void modify(int l, int r, T val) {
250        modify(0, 0, n - 1, l, r, val);
251    }
252
253    T query(int l, int r) {
254        return query(0, 0, n - 1, l, r);
255    }
256
257    T get(int pos) {
258        return query(pos, pos);
259    }
260
261    // Change clear() function to t.clear() if using
262    // unordered_map for SegTree!!!
263    void clear(int n_) {
264        n = n_;
265        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
266        lazy_mark;
267    }
268
269    void build(vector<T>& a) {
270        n = sz(a);
271        clear(n);
272        build(0, 0, n - 1, a);
273    }
274
275    void build(int v, int tl, int tr, vector<T>& a) {
276        if (tl == tr) {
277            t[v] = a[tl];
278            return;
279        }
280        int tm = (tl + tr) / 2;
281        // left child: [tl, tm]
282        // right child: [tm + 1, tr]
283        build(2 * v + 1, tl, tm, a);
284        build(2 * v + 2, tm + 1, tr, a);
285        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
286    }
287
288    LazySegTree(vector<T>& a) {
289        build(a);
290    }
291
292    void push(int v, int tl, int tr) {
293        if (lazy[v] == lazy_mark) return;
294        if (tl == tr) {
295            t[v] = lazy[v];
296            return;
297        }
298        push(v, tl, (tl + tr) / 2);
299        push(v, (tl + tr) / 2 + 1, tr);
300        t[v] = f(t[v], t[v]);
301    }
302
303    T query(int l, int r) {
304        return query(0, 0, n - 1, l, r);
305    }
306
307    void modify(int l, int r, T val) {
308        modify(0, 0, n - 1, l, r, val);
309    }
310
311    T query(int l, int r) {
312        return query(0, 0, n - 1, l, r);
313    }
314
315    T get(int pos) {
316        return query(pos, pos);
317    }
318
319    // Change clear() function to t.clear() if using
320    // unordered_map for SegTree!!!
321    void clear(int n_) {
322        n = n_;
323        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
324        lazy_mark;
325    }
326
327    void build(vector<T>& a) {
328        n = sz(a);
329        clear(n);
330        build(0, 0, n - 1, a);
331    }
332
333    void build(int v, int tl, int tr, vector<T>& a) {
334        if (tl == tr) {
335            t[v] = a[tl];
336            return;
337        }
338        int tm = (tl + tr) / 2;
339        // left child: [tl, tm]
340        // right child: [tm + 1, tr]
341        build(2 * v + 1, tl, tm, a);
342        build(2 * v + 2, tm + 1, tr, a);
343        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
344    }
345
346    LazySegTree(vector<T>& a) {
347        build(a);
348    }
349
350    void push(int v, int tl, int tr) {
351        if (lazy[v] == lazy_mark) return;
352        if (tl == tr) {
353            t[v] = lazy[v];
354            return;
355        }
356        push(v, tl, (tl + tr) / 2);
357        push(v, (tl + tr) / 2 + 1, tr);
358        t[v] = f(t[v], t[v]);
359    }
360
361    T query(int l, int r) {
362        return query(0, 0, n - 1, l, r);
363    }
364
365    void modify(int l, int r, T val) {
366        modify(0, 0, n - 1, l, r, val);
367    }
368
369    T query(int l, int r) {
370        return query(0, 0, n - 1, l, r);
371    }
372
373    T get(int pos) {
374        return query(pos, pos);
375    }
376
377    // Change clear() function to t.clear() if using
378    // unordered_map for SegTree!!!
379    void clear(int n_) {
380        n = n_;
381        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
382        lazy_mark;
383    }
384
385    void build(vector<T>& a) {
386        n = sz(a);
387        clear(n);
388        build(0, 0, n - 1, a);
389    }
390
391    void build(int v, int tl, int tr, vector<T>& a) {
392        if (tl == tr) {
393            t[v] = a[tl];
394            return;
395        }
396        int tm = (tl + tr) / 2;
397        // left child: [tl, tm]
398        // right child: [tm + 1, tr]
399        build(2 * v + 1, tl, tm, a);
400        build(2 * v + 2, tm + 1, tr, a);
401        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
402    }
403
404    LazySegTree(vector<T>& a) {
405        build(a);
406    }
407
408    void push(int v, int tl, int tr) {
409        if (lazy[v] == lazy_mark) return;
410        if (tl == tr) {
411            t[v] = lazy[v];
412            return;
413        }
414        push(v, tl, (tl + tr) / 2);
415        push(v, (tl + tr) / 2 + 1, tr);
416        t[v] = f(t[v], t[v]);
417    }
418
419    T query(int l, int r) {
420        return query(0, 0, n - 1, l, r);
421    }
422
423    void modify(int l, int r, T val) {
424        modify(0, 0, n - 1, l, r, val);
425    }
426
427    T query(int l, int r) {
428        return query(0, 0, n - 1, l, r);
429    }
430
431    T get(int pos) {
432        return query(pos, pos);
433    }
434
435    // Change clear() function to t.clear() if using
436    // unordered_map for SegTree!!!
437    void clear(int n_) {
438        n = n_;
439        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
440        lazy_mark;
441    }
442
443    void build(vector<T>& a) {
444        n = sz(a);
445        clear(n);
446        build(0, 0, n - 1, a);
447    }
448
449    void build(int v, int tl, int tr, vector<T>& a) {
450        if (tl == tr) {
451            t[v] = a[tl];
452            return;
453        }
454        int tm = (tl + tr) / 2;
455        // left child: [tl, tm]
456        // right child: [tm + 1, tr]
457        build(2 * v + 1, tl, tm, a);
458        build(2 * v + 2, tm + 1, tr, a);
459        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
460    }
461
462    LazySegTree(vector<T>& a) {
463        build(a);
464    }
465
466    void push(int v, int tl, int tr) {
467        if (lazy[v] == lazy_mark) return;
468        if (tl == tr) {
469            t[v] = lazy[v];
470            return;
471        }
472        push(v, tl, (tl + tr) / 2);
473        push(v, (tl + tr) / 2 + 1, tr);
474        t[v] = f(t[v], t[v]);
475    }
476
477    T query(int l, int r) {
478        return query(0, 0, n - 1, l, r);
479    }
480
481    void modify(int l, int r, T val) {
482        modify(0, 0, n - 1, l, r, val);
483    }
484
485    T query(int l, int r) {
486        return query(0, 0, n - 1, l, r);
487    }
488
489    T get(int pos) {
490        return query(pos, pos);
491    }
492
493    // Change clear() function to t.clear() if using
494    // unordered_map for SegTree!!!
495    void clear(int n_) {
496        n = n_;
497        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
498        lazy_mark;
499    }
500
501    void build(vector<T>& a) {
502        n = sz(a);
503        clear(n);
504        build(0, 0, n - 1, a);
505    }
506
507    void build(int v, int tl, int tr, vector<T>& a) {
508        if (tl == tr) {
509            t[v] = a[tl];
510            return;
511        }
512        int tm = (tl + tr) / 2;
513        // left child: [tl, tm]
514        // right child: [tm + 1, tr]
515        build(2 * v + 1, tl, tm, a);
516        build(2 * v + 2, tm + 1, tr, a);
517        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
518    }
519
520    LazySegTree(vector<T>& a) {
521        build(a);
522    }
523
524    void push(int v, int tl, int tr) {
525        if (lazy[v] == lazy_mark) return;
526        if (tl == tr) {
527            t[v] = lazy[v];
528            return;
529        }
530        push(v, tl, (tl + tr) / 2);
531        push(v, (tl + tr) / 2 + 1, tr);
532        t[v] = f(t[v], t[v]);
533    }
534
535    T query(int l, int r) {
536        return query(0, 0, n - 1, l, r);
537    }
538
539    void modify(int l, int r, T val) {
540        modify(0, 0, n - 1, l, r, val);
541    }
542
543    T query(int l, int r) {
544        return query(0, 0, n - 1, l, r);
545    }
546
547    T get(int pos) {
548        return query(pos, pos);
549    }
550
551    // Change clear() function to t.clear() if using
552    // unordered_map for SegTree!!!
553    void clear(int n_) {
554        n = n_;
555        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
556        lazy_mark;
557    }
558
559    void build(vector<T>& a) {
560        n = sz(a);
561        clear(n);
562        build(0, 0, n - 1, a);
563    }
564
565    void build(int v, int tl, int tr, vector<T>& a) {
566        if (tl == tr) {
567            t[v] = a[tl];
568            return;
569        }
570        int tm = (tl + tr) / 2;
571        // left child: [tl, tm]
572        // right child: [tm + 1, tr]
573        build(2 * v + 1, tl, tm, a);
574        build(2 * v + 2, tm + 1, tr, a);
575        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
576    }
577
578    LazySegTree(vector<T>& a) {
579        build(a);
580    }
581
582    void push(int v, int tl, int tr) {
583        if (lazy[v] == lazy_mark) return;
584        if (tl == tr) {
585            t[v] = lazy[v];
586            return;
587        }
588        push(v, tl, (tl + tr) / 2);
589        push(v, (tl + tr) / 2 + 1, tr);
590        t[v] = f(t[v], t[v]);
591    }
592
593    T query(int l, int r) {
594        return query(0, 0, n - 1, l, r);
595    }
596
597    void modify(int l, int r, T val) {
598        modify(0, 0, n - 1, l, r, val);
599    }
600
601    T query(int l, int r) {
602        return query(0, 0, n - 1, l, r);
603    }
604
605    T get(int pos) {
606        return query(pos, pos);
607    }
608
609    // Change clear() function to t.clear() if using
610    // unordered_map for SegTree!!!
611    void clear(int n_) {
612        n = n_;
613        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
614        lazy_mark;
615    }
616
617    void build(vector<T>& a) {
618        n = sz(a);
619        clear(n);
620        build(0, 0, n - 1, a);
621    }
622
623    void build(int v, int tl, int tr, vector<T>& a) {
624        if (tl == tr) {
625            t[v] = a[tl];
626            return;
627        }
628        int tm = (tl + tr) / 2;
629        // left child: [tl, tm]
630        // right child: [tm + 1, tr]
631        build(2 * v + 1, tl, tm, a);
632        build(2 * v + 2, tm + 1, tr, a);
633        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
634    }
635
636    LazySegTree(vector<T>& a) {
637        build(a);
638    }
639
640    void push(int v, int tl, int tr) {
641        if (lazy[v] == lazy_mark) return;
642        if (tl == tr) {
643            t[v] = lazy[v];
644            return;
645        }
646        push(v, tl, (tl + tr) / 2);
647        push(v, (tl + tr) / 2 + 1, tr);
648        t[v] = f(t[v], t[v]);
649    }
650
651    T query(int l, int r) {
652        return query(0, 0, n - 1, l, r);
653    }
654
655    void modify(int l, int r, T val) {
656        modify(0, 0, n - 1, l, r, val);
657    }
658
659    T query(int l, int r) {
660        return query(0, 0, n - 1, l, r);
661    }
662
663    T get(int pos) {
664        return query(pos, pos);
665    }
666
667    // Change clear() function to t.clear() if using
668    // unordered_map for SegTree!!!
669    void clear(int n_) {
670        n = n_;
671        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
672        lazy_mark;
673    }
674
675    void build(vector<T>& a) {
676        n = sz(a);
677        clear(n);
678        build(0, 0, n - 1, a);
679    }
680
681    void build(int v, int tl, int tr, vector<T>& a) {
682        if (tl == tr) {
683            t[v] = a[tl];
684            return;
685        }
686        int tm = (tl + tr) / 2;
687        // left child: [tl, tm]
688        // right child: [tm + 1, tr]
689        build(2 * v + 1, tl, tm, a);
690        build(2 * v + 2, tm + 1, tr, a);
691        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
692    }
693
694    LazySegTree(vector<T>& a) {
695        build(a);
696    }
697
698    void push(int v, int tl, int tr) {
699        if (lazy[v] == lazy_mark) return;
700        if (tl == tr) {
701            t[v] = lazy[v];
702            return;
703        }
704        push(v, tl, (tl + tr) / 2);
705        push(v, (tl + tr) / 2 + 1, tr);
706        t[v] = f(t[v], t[v]);
707    }
708
709    T query(int l, int r) {
710        return query(0, 0, n - 1, l, r);
711    }
712
713    void modify(int l, int r, T val) {
714        modify(0, 0, n - 1, l, r, val);
715    }
716
717    T query(int l, int r) {
718        return query(0, 0, n - 1, l, r);
719    }
720
721    T get(int pos) {
722        return query(pos, pos);
723    }
724
725    // Change clear() function to t.clear() if using
726    // unordered_map for SegTree!!!
727    void clear(int n_) {
728        n = n_;
729        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
730        lazy_mark;
731    }
732
733    void build(vector<T>& a) {
734        n = sz(a);
735        clear(n);
736        build(0, 0, n - 1, a);
737    }
738
739    void build(int v, int tl, int tr, vector<T>& a) {
740        if (tl == tr) {
741            t[v] = a[tl];
742            return;
743        }
744        int tm = (tl + tr) / 2;
745        // left child: [tl, tm]
746        // right child: [tm + 1, tr]
747        build(2 * v + 1, tl, tm, a);
748        build(2 * v + 2, tm + 1, tr, a);
749        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
750    }
751
752    LazySegTree(vector<T>& a) {
753        build(a);
754    }
755
756    void push(int v, int tl, int tr) {
757        if (lazy[v] == lazy_mark) return;
758        if (tl == tr) {
759            t[v] = lazy[v];
760            return;
761        }
762        push(v, tl, (tl + tr) / 2);
763        push(v, (tl + tr) / 2 + 1, tr);
764        t[v] = f(t[v], t[v]);
765    }
766
767    T query(int l, int r) {
768        return query(0, 0, n - 1, l, r);
769    }
770
771    void modify(int l, int r, T val) {
772        modify(0, 0, n - 1, l, r, val);
773    }
774
775    T query(int l, int r) {
776        return query(0, 0, n - 1, l, r);
777    }
778
779    T get(int pos) {
780        return query(pos, pos);
781    }
782
783    // Change clear() function to t.clear() if using
784    // unordered_map for SegTree!!!
785    void clear(int n_) {
786        n = n_;
787        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
788        lazy_mark;
789    }
790
791    void build(vector<T>& a) {
792        n = sz(a);
793        clear(n);
794        build(0, 0, n - 1, a);
795    }
796
797    void build(int v, int tl, int tr, vector<T>& a) {
798        if (tl == tr) {
799            t[v] = a[tl];
800            return;
801        }
802        int tm = (tl + tr) / 2;
803        // left child: [tl, tm]
804        // right child: [tm + 1, tr]
805        build(2 * v + 1, tl, tm, a);
806        build(2 * v + 2, tm + 1, tr, a);
807        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
808    }
809
810    LazySegTree(vector<T>& a) {
811        build(a);
812    }
813
814    void push(int v, int tl, int tr) {
815        if (lazy[v] == lazy_mark) return;
816        if (tl == tr) {
817            t[v] = lazy[v];
818            return;
819        }
820        push(v, tl, (tl + tr) / 2);
821        push(v, (tl + tr) / 2 + 1, tr);
822        t[v] = f(t[v], t[v]);
823    }
824
825    T query(int l, int r) {
826        return query(0, 0, n - 1, l, r);
827    }
828
829    void modify(int l, int r, T val) {
830        modify(0, 0, n - 1, l, r, val);
831    }
832
833    T query(int l, int r) {
834        return query(0, 0, n - 1, l, r);
835    }
836
837    T get(int pos) {
838        return query(pos, pos);
839    }
840
841    // Change clear() function to t.clear() if using
842    // unordered_map for SegTree!!!
843    void clear(int n_) {
844        n = n_;
845        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
846        lazy_mark;
847    }
848
849    void build(vector<T>& a) {
850        n = sz(a);
851        clear(n);
852        build(0, 0, n - 1, a);
853    }
854
855    void build(int v, int tl, int tr, vector<T>& a) {
856        if (tl == tr) {
857            t[v] = a[tl];
858            return;
859        }
860        int tm = (tl + tr) / 2;
861        // left child: [tl, tm]
862        // right child: [tm + 1, tr]
863        build(2 * v + 1, tl, tm, a);
864        build(2 * v + 2, tm + 1, tr, a);
865        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
866    }
867
868    LazySegTree(vector<T>& a) {
869        build(a);
870    }
871
872    void push(int v, int tl, int tr) {
873        if (lazy[v] == lazy_mark) return;
874        if (tl == tr) {
875            t[v] = lazy[v];
876            return;
877        }
878        push(v, tl, (tl + tr) / 2);
879        push(v, (tl + tr) / 2 + 1, tr);
880        t[v] = f(t[v], t[v]);
881    }
882
883    T query(int l, int r) {
884        return query(0, 0, n - 1, l, r);
885    }
886
887    void modify(int l, int r, T val) {
888        modify(0, 0, n - 1, l, r, val);
889    }
890
891    T query(int l, int r) {
892        return query(0, 0, n - 1, l, r);
893    }
894
895    T get(int pos) {
896        return query(pos, pos);
897    }
898
899    // Change clear() function to t.clear() if using
900    // unordered_map for SegTree!!!
901    void clear(int n_) {
902        n = n_;
903        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
904        lazy_mark;
905    }
906
907    void build(vector<T>& a) {
908        n = sz(a);
909        clear(n);
910        build(0, 0, n - 1, a);
911    }
912
913    void build(int v, int tl, int tr, vector<T>& a) {
914        if (tl == tr) {
915            t[v] = a[tl];
916            return;
917        }
918        int tm = (tl + tr) / 2;
919        // left child: [tl, tm]
920        // right child: [tm + 1, tr]
921        build(2 * v + 1, tl, tm, a);
922        build(2 * v + 2, tm + 1, tr, a);
923        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
924    }
925
926    LazySegTree(vector<T>& a) {
927        build(a);
928    }
929
930    void push(int v, int tl, int tr) {
931        if (lazy[v] == lazy_mark) return;
932        if (tl == tr) {
933            t[v] = lazy[v];
934            return;
935        }
936        push(v, tl, (tl + tr) / 2);
937        push(v, (tl + tr) / 2 + 1, tr);
938        t[v] = f(t[v], t[v]);
939    }
940
941    T query(int l, int r) {
942        return query(0, 0, n - 1, l, r);
943    }
944
945    void modify(int l, int r, T val) {
946        modify(0, 0, n - 1, l, r, val);
947    }
948
949    T query(int l, int r) {
950        return query(0, 0, n - 1, l, r);
951    }
952
953    T get(int pos) {
954        return query(pos, pos);
955    }
956
957    // Change clear() function to t.clear() if using
958    // unordered_map for SegTree!!!
959    void clear(int n_) {
960        n = n_;
961        for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
962        lazy_mark;
963    }
964
965    void build(vector<T>& a) {
966        n = sz(a);
967        clear(n);
968        build(0, 0, n - 1, a);
969    }
970
971    void build(int v, int tl, int tr, vector<T>& a) {
972        if (tl == tr) {
973            t[v] = a[tl];
974            return;
975        }
976        int tm = (tl + tr) / 2;
977        // left child: [tl, tm]
978        // right child: [tm + 1, tr]
979        build(2 * v + 1, tl, tm, a);
980        build(2 * v + 2, tm + 1, tr, a);
981        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
982    }
983
984    LazySegTree(vector<T>& a) {
985        build(a);
986    }
987
988    void push(int v, int tl, int tr) {
989        if (lazy[v] == lazy_mark) return;
990        if (tl == tr) {
991            t[v] = lazy[v];
992            return;
993        }
994        push(v, tl, (tl + tr) / 2);
995        push(v, (tl +
```

```

15   lg[1] = 0;
16   for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
17
18   for (int k = 0; k < LOG; k++){
19       for (int i = 0; i < n; i++){
20           if (!k) st[i][k] = a[i];
21           else st[i][k] = f(st[i][k - 1], st[min(n - 1, i + (1 <<
↪ (k - 1))))[k - 1]);
22       }
23   }
24 }
25
26 T query(int l, int r){
27     int sz = r - l + 1;
28     return f(st[l][lg[sz]], st[r - (1 << lg[sz]) + 1][lg[sz]]);
29 }
30 };

```

Suffix Array and LCP array

- (uses SparseTable above)

```

1 struct SuffixArray{
2     vector<int> p, c, h;
3     SparseTable<int> st;
4     /*
5      * In the end, array c gives the position of each suffix in p
6      * using 1-based indexation!
7      */
8
9     SuffixArray() {}
10
11     SuffixArray(string s){
12         buildArray(s);
13         buildLCP(s);
14         buildSparse();
15     }
16
17     void buildArray(string s){
18         int n = sz(s) + 1;
19         p.resize(n), c.resize(n);
20         for (int i = 0; i < n; i++) p[i] = i;
21         sort(all(p), [&] (int a, int b){return s[a] < s[b];});
22         c[p[0]] = 0;
23         for (int i = 1; i < n; i++){
24             c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
25         }
26         vector<int> p2(n), c2(n);
27         // w is half-length of each string.
28         for (int w = 1; w < n; w <= 1){
29             for (int i = 0; i < n; i++){
30                 p2[i] = (p[i] - w + n) % n;
31             }
32             vector<int> cnt(n);
33             for (auto i : c) cnt[i]++;
34             for (int i = 1; i < n; i++) cnt[i] += cnt[i - 1];
35             for (int i = n - 1; i >= 0; i--){
36                 p[--cnt[c[p2[i]]]] = p2[i];
37             }
38             c2[p[0]] = 0;
39             for (int i = 1; i < n; i++){
40                 c2[p[i]] = c2[p[i - 1]] +
41                 (c[p[i]] != c[p[i - 1]] ||
42                 c[(p[i] + w) % n] != c[(p[i - 1] + w) % n]);
43             }
44             c.swap(c2);
45         }
46         p.erase(p.begin());
47     }
48
49     void buildLCP(string s){
50         // The algorithm assumes that suffix array is already
51         ↪ built on the same string.
52         int n = sz(s);
53         h.resize(n - 1);
54         int k = 0;
55         for (int i = 0; i < n; i++){

```

```

55         if (c[i] == n){
56             k = 0;
57             continue;
58         }
59         int j = p[c[i]];
60         while (i + k < n && j + k < n && s[i + k] == s[j + k])
↪ k++;
61         h[c[i] - 1] = k;
62         if (k) k--;
63     }
64     /*
65     * Then an RMQ Sparse Table can be built on array h
66     * to calculate LCP of 2 non-consecutive suffixes.
67     */
68 }
69
70 void buildSparse(){
71     st.build(h);
72 }
73
74 // l and r must be in 0-BASED INDEXATION
75 int lcp(int l, int r){
76     l = c[l] - 1, r = c[r] - 1;
77     if (l > r) swap(l, r);
78     return st.query(l, r - 1);
79 }
80 };

```

Aho Corasick Trie

- For each node in the trie, the suffix link points to the longest proper suffix of the represented string. The terminal-link tree has square-root height (can be constructed by DFS).

```

1 const int S = 26;
2
3 // Function converting char to int.
4 int ctoi(char c){
5     return c - 'a';
6 }
7
8 // To add terminal links, use DFS
9 struct Node{
10     vector<int> nxt;
11     int link;
12     bool terminal;
13
14     Node() {
15         nxt.assign(S, -1), link = 0, terminal = 0;
16     }
17 };
18
19 vector<Node> trie(1);
20
21 // add_string returns the terminal vertex.
22 int add_string(string& s){
23     int v = 0;
24     for (auto c : s){
25         int cur = ctoi(c);
26         if (trie[v].nxt[cur] == -1){
27             trie[v].nxt[cur] = sz(trie);
28             trie.emplace_back();
29         }
30         v = trie[v].nxt[cur];
31     }
32     trie[v].terminal = 1;
33     return v;
34 }
35
36 /*
37 * Suffix links are compressed.
38 * This means that:
39 * If vertex v has a child by letter x, then:
40 *   trie[v].nxt[x] points to that child.
41 * If vertex v doesn't have such child, then:

```

```

42     trie[v].nxt[x] points to the suffix link of that child
43     if we would actually have it.
44 */
45 void add_links(){
46     queue<int> q;
47     q.push(0);
48     while (!q.empty()){
49         auto v = q.front();
50         int u = trie[v].link;
51         q.pop();
52         for (int i = 0; i < S; i++){
53             int& ch = trie[v].nxt[i];
54             if (ch == -1){
55                 ch = v? trie[u].nxt[i] : 0;
56             }
57             else{
58                 trie[ch].link = v? trie[u].nxt[i] : 0;
59                 q.push(ch);
60             }
61         }
62     }
63 }
64
65 bool is_terminal(int v){
66     return trie[v].terminal;
67 }
68
69 int get_link(int v){
70     return trie[v].link;
71 }
72
73 int go(int v, char c){
74     return trie[v].nxt[ctoi(c)];
75 }

```

Convex Hull Trick

- Allows to insert a linear function to the hull in $O(1)$ and get the minimum/maximum value of the stored function at a point in $O(\log n)$.
- NOTE: The lines must be added in the order of decreasing/increasing gradients. CAREFULLY CHECK THE SETUP BEFORE USING!
- IMPORTANT: THE DEFAULT VERSION SURELY WORKS. IF MODIFIED VERSIONS DON'T WORK, TRY TRANSFORMING THEM TO THE DEFAULT ONE BY CHANGING SIGNS.

```

1 struct line{
2     ll k, b;
3     ll f(ll x){
4         return k * x + b;
5     };
6 };
7
8 vector<line> hull;
9
10 void add_line(line nl){
11     if (!hull.empty() && hull.back().k == nl.k){
12         nl.b = min(nl.b, hull.back().b); // Default: minimum. For
13         ↪ maximum change "min" to "max".
14         hull.pop_back();
15     }
16     while (sz(hull) > 1){
17         auto& l1 = hull.end()[-2], l2 = hull.back();
18         if ((nl.b - l1.b) * (l2.k - nl.k) >= (nl.b - l2.b) * (l1.k
19         ↪ - nl.k)) hull.pop_back(); // Default: decreasing gradient
20         ↪ k. For increasing k change the sign to <=.
21         else break;
22     }
23     hull.pb(nl);
24 }
25
26 ll get(ll x){

```

```

24     int l = 0, r = sz(hull);
25     while (r - l > 1){
26         int mid = (l + r) / 2;
27         if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid; //
28         ↪ Default: minimum. For maximum change the sign to <=.
29         else r = mid;
30     }
31     return hull[l].f(x);
32 }

```

Li-Chao Segment Tree

- allows to add linear functions in any order and query minimum/maximum value of those at a point, all in $O(\log n)$.
- Clear: clear()

```

1 const ll INF = 1e18; // Change the constant!
2 struct LiChaoTree{
3     struct line{
4         ll k, b;
5         line(){
6             k = b = 0;
7         };
8         line(ll k_, ll b_){
9             k = k_, b = b_;
10        };
11        ll f(ll x){
12            return k * x + b;
13        };
14    };
15    int n;
16    bool minimum, on_points;
17    vector<ll> pts;
18    vector<line> t;
19
20    void clear(){
21        for (auto& l : t) l.k = 0, l.b = minimum? INF : -INF;
22    }
23
24    LiChaoTree(int n_, bool min_){ // This is a default
25        ↪ constructor for numbers in range [0, n - 1].
26        n = n_, minimum = min_, on_points = false;
27        t.resize(4 * n);
28        clear();
29    };
30
31    LiChaoTree(vector<ll> pts_, bool min_){ // This constructor
32        ↪ will build LCT on the set of points you pass. The points
33        ↪ may be in any order and contain duplicates.
34        pts = pts_, minimum = min_;
35        sort(all(pts));
36        pts.erase(unique(all(pts)), pts.end());
37        on_points = true;
38        n = sz(pts);
39        t.resize(4 * n);
40        clear();
41    };
42
43    void add_line(int v, int l, int r, line nl){
44        ↪ Adding on segment [l, r)
45        int m = (l + r) / 2;
46        ll lval = on_points? pts[l] : l, mval = on_points? pts[m]
47        ↪ : m;
48        if ((minimum && nl.f(mval) < t[v].f(mval)) || (!minimum &&
49        ↪ nl.f(mval) > t[v].f(mval))) swap(t[v], nl);
50        if (r - l == 1) return;
51        if ((minimum && nl.f(lval) < t[v].f(lval)) || (!minimum &&
52        ↪ nl.f(lval) > t[v].f(lval))) add_line(2 * v + 1, l, m, nl);
53        else add_line(2 * v + 2, m, r, nl);
54    }
55
56    ll get(int v, int l, int r, int x){
57        int m = (l + r) / 2;
58        if (r - l == 1) return t[v].f(on_points? pts[x] : x);
59        else{

```

```

54         if (minimum) return min(t[v].f(on_points? pts[x] : x), x
↪ < m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
55         else return max(t[v].f(on_points? pts[x] : x), x < m?
↪ get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
56     }
57 }
58
59 void add_line(ll k, ll b){
60     add_line(0, 0, n, line(k, b));
61 }
62
63 ll get(ll x){
64     return get(0, 0, n, on_points? lower_bound(all(pts), x) -
↪ pts.begin() : x);
65 }; // Always pass the actual value of x, even if LCT is on
↪ points.
66 };

```

Persistent Segment Tree

- for RSQ

```

1  struct Node {
2      ll val;
3      Node *l, *r;
4
5      Node(ll x) : val(x), l(nullptr), r(nullptr) {}
6      Node(Node *ll, Node *rr) {
7          l = ll, r = rr;
8          val = 0;
9          if (l) val += l->val;
10         if (r) val += r->val;
11     }
12     Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
13 };
14 const int N = 2e5 + 20;
15 ll a[N];
16 Node *roots[N];
17 int n, cnt = 1;
18 Node *build(int l = 1, int r = n) {
19     if (l == r) return new Node(a[l]);
20     int mid = (l + r) / 2;
21     return new Node(build(l, mid), build(mid + 1, r));
22 }
23 Node *update(Node *node, int val, int pos, int l = 1, int r =
↪ n) {
24     if (l == r) return new Node(val);
25     int mid = (l + r) / 2;
26     if (pos > mid)
27         return new Node(node->l, update(node->r, val, pos, mid
↪ + 1, r));
28     else return new Node(update(node->l, val, pos, l, mid),
↪ node->r);
29 }
30 ll query(Node *node, int a, int b, int l = 1, int r = n) {
31     if (l > b || r < a) return 0;
32     if (l >= a && r <= b) return node->val;
33     int mid = (l + r) / 2;
34     return query(node->l, a, b, l, mid) + query(node->r, a, b,
↪ mid + 1, r);
35 }

```

Miscellaneous

Ordered Set

```

1  #include <ext/pb_ds/assoc_container.hpp>
2  #include <ext/pb_ds/tree_policy.hpp>
3  using namespace __gnu_pbds;
4  typedef tree<int, null_type, less<int>, rb_tree_tag,
↪ tree_order_statistics_node_update> ordered_set;

```

Measuring Execution Time

```

1  ld tic = clock();
2  // execute algo...
3  ld tac = clock();
4  // Time in milliseconds
5  cerr << (tac - tic) / CLOCKS_PER_SEC * 1000 << endl;
6  // No need to comment out the print because it's done to cerr.

```

Setting Fixed D.P. Precision

```

1  cout << setprecision(d) << fixed;
2  // Each number is rounded to d digits after the decimal point,
↪ and truncated.

```

Common Bugs and General Advice

- Check overflow, array bounds
- Check variable overloading
- Check special cases (n=1?)
- Do something instead of nothing, stay organized
- Write stuff down!
- Don't get stuck on one approach!