

# 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
Extended Euclidean Algorithm . . . . .	8
Linear Sieve . . . . .	8
Gaussian Elimination . . . . .	8
NTT . . . . .	9
FFT . . . . .	9
is_prime . . . . .	9
Berlekamp-Massey . . . . .	10

## Data Structures

Fenwick Tree . . . . .	10
Lazy Propagation SegTree . . . . .	10
Sparse Table . . . . .	11
Suffix Array and LCP array . . . . .	11
Aho Corasick Trie . . . . .	12
Convex Hull Trick . . . . .	12
Li-Chao Segment Tree . . . . .	13
Persistent Segment Tree . . . . .	13

## Miscellaneous

Ordered Set . . . . .	13
Measuring Execution Time . . . . .	13
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     long long cap, flow = 0;
4     FlowEdge(int v, int u, long long cap) : v(v), u(u),
5     cap(cap) {}
6 };
7 struct Dinic {
8     const long long flow_inf = 1e18;
9     vector<FlowEdge> edges;
10    vector<vector<int>> adj;
11    int n, m = 0;
12    int s, t;
13    vector<int> level, ptr;
14    queue<int> q;
15    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
16        adj.resize(n);
17        level.resize(n);
18        ptr.resize(n);
19    }
20    void add_edge(int v, int u, long long cap) {
21        edges.emplace_back(v, u, cap);
22        edges.emplace_back(u, v, 0);
23        adj[v].push_back(m);
24        adj[u].push_back(m + 1);
25        m += 2;
26    }
27    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 long long dfs(int v, long long 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         long long 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 long long flow() {
64     long long 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 (long long 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),
26 ↪ its(n), pe(n) {}
27
28 int add(int from, int to, T forward_cap, C edge_cost, T
29 ↪ backward_cap = 0) {
30     assert(0 <= from && from < n && 0 <= to && to < n);
31     assert(forward_cap >= 0 && backward_cap >= 0);
32     int id = static_cast<int>(edges.size());
33     g[from].push_back(id);
34     edges.push_back({from, to, forward_cap, 0, edge_cost});
35     g[to].push_back(id + 1);
36     edges.push_back({to, from, backward_cap, 0, -edge_cost});
37     return id;
38 }
39
40 void expath(int st) {
41     fill(d.begin(), d.end(), INF_C);
42     q.clear();
43     fill(its.begin(), its.end(), q.end());
44     its[st] = q.push({pot[st], st});
45     d[st] = 0;
46     while (!q.empty()) {
47         int i = q.top().second;
48         q.pop();
49         its[i] = q.end();
50         for (int id : g[i]) {
51             const edge &e = edges[id];
52             int j = e.to;
53             if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
54                 d[j] = d[i] + e.cost;
55                 pe[j] = id;
56                 if (its[j] == q.end()) {
57                     its[j] = q.push({pot[j] - d[j], j});
58                 } else {
59                     q.modify(its[j], {pot[j] - d[j], j});
60                 }
61             }
62         }
63     }
64     swap(d, pot);
65 }
66
67 pair<T, C> max_flow(int st, int fin) {
68     T flow = 0;
69     C cost = 0;
70     bool ok = true;
71     for (auto& e : edges) {
72         if (e.c - e.f > eps && e.cost + pot[e.from] - pot[e.to]
73         ↪ < 0) {
74             ok = false;
75             break;
76         }
77     }
78     if (ok) {
79         expath(st);
80     } else {
81         vector<int> deg(n, 0);
82         for (int i = 0; i < n; i++) {
83             for (int eid : g[i]) {
84                 auto& e = edges[eid];
85                 if (e.c - e.f > eps) {
86                     deg[e.to] += 1;
87                 }
88             }
89         }
90         vector<int> que;
91         for (int i = 0; i < n; i++) {
92             if (deg[i] == 0) {
93                 que.push_back(i);
94             }
95         }

```

```

96         for (int b = 0; b < (int) que.size(); b++) {
97             for (int eid : g[que[b]]) {
98                 auto& e = edges[eid];
99                 if (e.c - e.f > eps) {
100                     deg[e.to] -= 1;
101                     if (deg[e.to] == 0) {
102                         que.push_back(e.to);
103                     }
104                 }
105             }
106         }
107     }
108     fill(pot.begin(), pot.end(), INF_C);
109     pot[st] = 0;
110     if (static_cast<int>(que.size()) == n) {
111         for (int v : que) {
112             if (pot[v] < INF_C) {
113                 for (int eid : g[v]) {
114                     auto& e = edges[eid];
115                     if (e.c - e.f > eps) {
116                         if (pot[v] + e.cost < pot[e.to]) {
117                             pot[e.to] = pot[v] + e.cost;
118                             pe[e.to] = eid;
119                         }
120                     }
121                 }
122             }
123         }
124     } else {
125         que.assign(1, st);
126         vector<bool> in_queue(n, false);
127         in_queue[st] = true;
128         for (int b = 0; b < (int) que.size(); b++) {
129             int i = que[b];
130             in_queue[i] = false;
131             for (int id : g[i]) {
132                 const edge &e = edges[id];
133                 if (e.c - e.f > eps && pot[i] + e.cost <
134                 ↪ pot[e.to]) {
135                     pot[e.to] = pot[i] + e.cost;
136                     pe[e.to] = id;
137                     if (!in_queue[e.to]) {
138                         que.push_back(e.to);
139                         in_queue[e.to] = true;
140                     }
141                 }
142             }
143         }
144     }
145     while (pot[fin] < INF_C) {
146         T push = numeric_limits<T>::max();
147         int v = fin;
148         while (v != st) {
149             const edge &e = edges[pe[v]];
150             push = min(push, e.c - e.f);
151             v = e.from;
152         }
153         v = fin;
154         while (v != st) {
155             edge &e = edges[pe[v]];
156             e.f += push;
157             edge &back = edges[pe[v] ^ 1];
158             back.f -= push;
159             v = e.from;
160         }
161         flow += push;
162         cost += push * pot[fin];
163         expath(st);
164     }
165     return {flow, cost};
166 }
167 };
168
169 // Examples: MCMF<int, int> g(n); g.add(u,v,c,w,0);
170 ↪ g.max_flow(s,t).
171 // To recover flow through original edges: iterate over even
172 ↪ 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     } while (j1 != j0);
20     for (int j=0; j<=m; ++j)
21         if (used[j])
22             u[p[j]] += delta, v[j] -= delta;
23     else
24         minv[j] -= delta;
25     j0 = j1;
26 } while (p[j0] != 0);
27 do {
28     int j1 = way[j0];
29     p[j0] = p[j1];
30     j0 = j1;
31 } while (j0);
32 }
33 vector<int> ans (n+1); // ans[i] stores the column selected
34 ↪ for row i
35 for (int j=1; j<=m; ++j)
36     ans[p[j]] = j;
37 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 }

```

### 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 = acosl(-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 /= 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 }

```

```

20 ll get_max_factor(ll _x) {
21     ll max_factor = 0;
22     function<void(ll)> fac = [&](ll x) {
23         if (x <= max_factor || x < 2) return;
24         if (is_prime(x)) {
25             max_factor = max_factor > x ? max_factor : x;
26             return;
27         }
28         ll p = x;
29         while (p >= x) p = pollard_rho(x);
30         while ((x % p) == 0) x /= p;
31         fac(x), fac(p);
32     };
33     fac(_x);
34     return max_factor;
35 }

```

## 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]) %
        MOD;
8         if (d == 0) continue;
9         vector<ll> temp = c;
10        ll coef = d * power(ldd, MOD - 2) % MOD;
11        for (int j = m; j < n; j++){
12            c[j] = (c[j] + MOD - coef * b[j - m]) % MOD;
13            if (c[j] < 0) c[j] += MOD;
14        }
15        if (2 * l <= i) {
16            l = i + 1 - l;
17            b = temp;
18            ldd = d;
19            m = 0;
20        }
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 }

```

## 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
    ↪ propagation is needed.
12    function<T(T, T)> f = [&] (T a, T b){
13        return a + b;
14    };
15    // f_on_seg calculates the function f, knowing the lazy
    ↪ value on segment,
16    // segment's size and the previous value.
17    // The default is segment modification for RSQ. For
    ↪ increments change to:
18    // return cur_seg_val + seg_size * lazy_val;
19    // For RMQ. Modification: return lazy_val; Increments:
    ↪ return cur_seg_val + lazy_val;
20    function<T(T, int, T)> f_on_seg = [&] (T cur_seg_val, int
    ↪ seg_size, T lazy_val){
21        return seg_size * lazy_val;
22    };
23    // upd_lazy updates the value to be propagated to child
    ↪ segments.
24    // Default: modification. For increments change to:
25    // lazy[v] = (lazy[v] == lazy_mark? val : lazy[v] +
    ↪ val);
26    function<void(int, T)> upd_lazy = [&] (int v, T val){
27        lazy[v] = val;
28    };
29    // Tip: for "get element on single index" queries, use max()
    ↪ on segment: no overflows.
30
31    LazySegTree(int n_) : n(n_) {
32        clear(n);
33    }
34
35    void build(int v, int tl, int tr, vector<T>& a){
36        if (tl == tr) {
37            t[v] = a[tl];
38            return;
39        }
40        int tm = (tl + tr) / 2;
41        // left child: [tl, tm]
42        // right child: [tm + 1, tr]
43        build(2 * v + 1, tl, tm, a);
44        build(2 * v + 2, tm + 1, tr, a);
45        t[v] = f(t[2 * v + 1], t[2 * v + 2]);
46    }
47
48    LazySegTree(vector<T>& a){
49        build(a);
50    }
51
52    void push(int v, int tl, int tr){
53        if (lazy[v] == lazy_mark) return;
54        int tm = (tl + tr) / 2;
55        t[2 * v + 1] = f_on_seg(t[2 * v + 1], tm - tl + 1,
    ↪ lazy[v]);
56        t[2 * v + 2] = f_on_seg(t[2 * v + 2], tr - tm, lazy[v]);
57        upd_lazy(2 * v + 1, lazy[v]), upd_lazy(2 * v + 2,
    ↪ lazy[v]);
58        lazy[v] = lazy_mark;
59    }
60
61    void modify(int v, int tl, int tr, int l, int r, T val){
62        if (l > r) return;
63        if (tl == l && tr == r){
64            t[v] = f_on_seg(t[v], tr - tl + 1, val);
65            upd_lazy(v, val);

```

```

66     return;
67 }
68 push(v, tl, tr);
69 int tm = (tl + tr) / 2;
70 modify(2 * v + 1, tl, tm, l, min(r, tm), val);
71 modify(2 * v + 2, tm + 1, tr, max(l, tm + 1), r, val);
72 t[v] = f(t[2 * v + 1], t[2 * v + 2]);
73 }
74
75 T query(int v, int tl, int tr, int l, int r) {
76     if (l > r) return default_return;
77     if (tl == l && tr == r) return t[v];
78     push(v, tl, tr);
79     int tm = (tl + tr) / 2;
80     return f(
81         query(2 * v + 1, tl, tm, l, min(r, tm)),
82         query(2 * v + 2, tm + 1, tr, max(l, tm + 1), r)
83     );
84 }
85
86 void modify(int l, int r, T val){
87     modify(0, 0, n - 1, l, r, val);
88 }
89
90 T query(int l, int r){
91     return query(0, 0, n - 1, l, r);
92 }
93
94 T get(int pos){
95     return query(pos, pos);
96 }
97
98 // Change clear() function to t.clear() if using
99 ⇨ unordered_map for SegTree!!!
100 void clear(int n_){
101     n = n_;
102     for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
103     ⇨ lazy_mark;
104 }
105
106 void build(vector<T>& a){
107     n = sz(a);
108     clear(n);
109     build(0, 0, n - 1, a);
110 }
111
112 };

```

## Sparse Table

```

1  const int N = 2e5 + 10, LOG = 20; // Change the constant!
2  template<typename T>
3  struct SparseTable{
4      int lg[N];
5      T st[N][LOG];
6      int n;
7
8      // Change this function
9      function<T(T, T)> f = [&] (T a, T b){
10         return min(a, b);
11     };
12
13     void build(vector<T>& a){
14         n = sz(a);
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 <<
22                 ⇨ (k - 1))))[k - 1]);
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++){
56             if (c[i] == n){
57                 k = 0;
58                 continue;
59             }
60             int j = p[c[i]];
61             while (i + k < n && j + k < n && s[i + k] == s[j + k])
62             ⇨ k++;
63             h[c[i] - 1] = k;
64             if (k) k--;
65         }
66         /*
67         Then an RMQ Sparse Table can be built on array h
68         to calculate LCP of 2 non-consecutive suffixes.
69         */

```

```

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 bool is_terminal(int v){
65     return trie[v].terminal;
66 }
67
68 int get_link(int v){
69     return trie[v].link;
70 }
71
72 int go(int v, char c){
73     return trie[v].nxt[ctoi(c)];
74 }
75

```

## Convex Hull Trick

- Allows to insert a linear function to the hull in (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){
27     int l = 0, r = sz(hull);
28     while (r - l > 1){
29         int mid = (l + r) / 2;
30         if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid; //
31         ↪ Default: minimum. For maximum change the sign to <=.
32         else r = mid;
33     }
34     return hull[l].f(x);
35 }

```

## 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{
60             if (minimum) return min(t[v].f(on_points? pts[x] : x), x
61             < m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
62             else return max(t[v].f(on_points? pts[x] : x), x < m?
63             get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
64         }
65     }
66
67     void add_line(ll k, ll b){
68         add_line(0, 0, n, line(k, b));
69     }
70 }
```

```
63     ll get(ll x){
64         return get(0, 0, n, on_points? lower_bound(all(pts), x) -
65         pts.begin() : x);
66     }; // Always pass the actual value of x, even if LCT is on
67     // points.
68 }
```

## 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 =
24     n) {
25     if (l == r) return new Node(val);
26     int mid = (l + r) / 2;
27     if (pos > mid)
28         return new Node(node->l, update(node->r, val, pos, mid
29         + 1, r));
30     else return new Node(update(node->l, val, pos, l, mid),
31     node->r);
32 }
33 ll query(Node *node, int a, int b, int l = 1, int r = n) {
34     if (l > b || r < a) return 0;
35     if (l >= a && r <= b) return node->val;
36     int mid = (l + r) / 2;
37     return query(node->l, a, b, l, mid) + query(node->r, a, b,
38     mid + 1, r);
39 }
```

## 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,
5  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!