

# The Reference

## Contents

### 1 data structures

1.1	Ordered Set Gnu Pbd	1
1.2	Segtree Rmq Lazy Max Update	1
1.3	Segtree Rmq Lazy Range	2
1.4	Segtree Point Rmq	2
1.5	Segtree Rsq Lazy Range Sum	2
1.6	Segtree Rxq Lazy Range Xor	3
1.7	Dsu	4
1.8	Dsu	4
1.9	Sparse Table Rmq	4

### 2 graphs

2.1	Scc-nodes-(kosajaru)	4
2.2	2-sat-(struct)	5
2.3	Floyd Warshall	5
2.4	Topological-sorting	6
2.5	Lowest Common Ancestor Sparse Table	6
2.6	Count-scc-(kosajaru)	6
2.7	Kruskal	7
2.8	Scc-(struct)	7
2.9	Check-bipartite	8
2.10	Dijkstra	8

### 3 extras

3.1	Binary To Gray	8
3.2	Bigint	8
3.3	Get-permutation-cycles	11

### 4 dynamic programming

4.1	Edit Distance	11
4.2	Money Sum Bottom Up	12
4.3	Knapsack Dp Values 01	12
4.4	Tsp	12

### 5 trees

5.1	Binary-lifting	12
5.2	Maximum-distances	12
5.3	Tree Diameter	13

### 6 searching

6.1	Ternary Search Recursive	14
-----	--------------------------	----

### 7 math

7.1	Power-sum	14
7.2	Sieve-list-primes	14
7.3	Factorial	14
7.4	Permutation-count	14
7.5	N-choose-k-count	14
7.6	Gcd-using-factorization	15
7.7	Is-prime	15
7.8	Fast Exp	15
7.9	Lcm-using-factorization	15
7.10	Euler-phi	15
7.11	Polynomial	16
7.12	Integer Mod	16
7.13	Count Divisors Memo	16
7.14	Lcm	17
7.15	Factorial-factorization	17
7.16	Factorization-with-primes	17
7.17	Modular-inverse-using-phi	17
7.18	Factorization	18
7.19	Gcd	18
7.20	Combinatorics With Repetitions	18

### 8 strings

8.1	Rabin-karp	18
8.2	Trie-naive	18
8.3	String-psum	19

# 1 data structures

## 1.1 Ordered Set Gnu Pbds

```
1 #include <ext/pb_ds/assoc_container.hpp>
2 #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T>
5 // using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
6 // tree_order_statistics_node_update>;
7
8 // if you want to find the elements less or equal :p
9 using ordered_set = tree<T, null_type, less_equal<T>, rb_tree_tag,
10 tree_order_statistics_node_update>;
```

## 1.2 Segtree Rmq Lazy Max Update

```
1 struct SegmentTree {
2     int N;
3     vll ns, lazy;
4     SegmentTree(const vll &xs) : N(xs.size()), ns(4 * N, 0), lazy(4 * N, 0) {
5         for (size_t i = 0; i < xs.size(); ++i) {
6             update(i, i, xs[i]);
7         }
8     }
9     void update(int a, int b, ll value) { update(1, 0, N - 1, a, b, value); }
10    void update(int node, int L, int R, int a, int b, ll value) {
11        if (lazy[node]) {
12            ns[node] = max(ns[node], lazy[node]);
13            if (L < R) {
14                lazy[2 * node] = max(lazy[2 * node], lazy[node]);
15                lazy[2 * node + 1] = max(lazy[2 * node + 1], lazy[node]);
16            }
17            lazy[node] = 0;
18        }
19        if (a > R or b < L) return;
20        if (a <= L and R <= b) {
21            ns[node] = max(ns[node], value);
22            if (L < R) {
23                lazy[2 * node] = max(value, lazy[2 * node]);
24                lazy[2 * node + 1] = max(value, lazy[2 * node + 1]);
25            }
26            return;
27        }
28        update(2 * node, L, (L + R) / 2, a, b, value);
29        update(2 * node + 1, (L + R) / 2 + 1, R, a, b, value);
30        ns[node] = max(ns[2 * node], ns[2 * node + 1]);
31    }
32
33    ll RMQ(int a, int b) { return RMQ(1, 0, N - 1, a, b); }
34    ll RMQ(int node, int L, int R, int a, int b) {
35        if (lazy[node]) {
36            ns[node] = max(ns[node], lazy[node]);
37            if (L < R) {
38                lazy[2 * node] = max(lazy[2 * node], lazy[node]);
39                lazy[2 * node + 1] = max(lazy[2 * node + 1], lazy[node]);
40            }
41        }
```

```
41        lazy[node] = 0;
42    }
43
44    if (a > R or b < L) return 0;
45    if (a <= L and R <= b) return ns[node];
46    ll x = RMQ(2 * node, L, (L + R) / 2, a, b);
47    ll y = RMQ(2 * node + 1, (L + R) / 2 + 1, R, a, b);
48    return max(x, y);
49 }
50 };
```

## 1.3 Segtree Rmq Lazy Range

```
1 struct SegmentTree {
2     int N;
3     vll ns, lazy;
4     SegmentTree(const vll &xs)
5         : N(xs.size()), ns(4 * N, INT_MAX), lazy(4 * N, 0) {
6         for (size_t i = 0; i < xs.size(); ++i) update(i, i, xs[i]);
7     }
8     void update(int a, int b, ll value) { update(1, 0, N - 1, a, b, value); }
9     void update(int node, int L, int R, int a, int b, ll value) {
10        if (lazy[node]) {
11            ns[node] = ns[node] == INT_MAX ? lazy[node] : ns[node] + lazy[node];
12            if (L < R) {
13                lazy[2 * node] += lazy[node];
14                lazy[2 * node + 1] += lazy[node];
15            }
16            lazy[node] = 0;
17        }
18        if (a > R or b < L) return;
19        if (a <= L and R <= b) {
20            ns[node] = ns[node] == INT_MAX ? value : ns[node] + value;
21            if (L < R) {
22                lazy[2 * node] += value;
23                lazy[2 * node + 1] += value;
24            }
25            return;
26        }
27        update(2 * node, L, (L + R) / 2, a, b, value);
28        update(2 * node + 1, (L + R) / 2 + 1, R, a, b, value);
29        ns[node] = min(ns[2 * node], ns[2 * node + 1]);
30    }
31    ll RMQ(int a, int b) { return RMQ(1, 0, N - 1, a, b); }
32    ll RMQ(int node, int L, int R, int a, int b) {
33        if (lazy[node]) {
34            ns[node] = ns[node] == INT_MAX ? lazy[node] : ns[node] + lazy[node];
35            if (L < R) {
36                lazy[2 * node] += lazy[node];
37                lazy[2 * node + 1] += lazy[node];
38            }
39            lazy[node] = 0;
40        }
41        if (a > R or b < L) return INT_MAX;
42
43        if (a <= L and R <= b) return ns[node];
44        ll x = RMQ(2 * node, L, (L + R) / 2, a, b);
45        ll y = RMQ(2 * node + 1, (L + R) / 2 + 1, R, a, b);
```

```

46     return min(x, y);
47 }
48 };

```

## 1.4 Segtree Point Rmq

```

1 class SegTree {
2 public:
3     int n;
4     vector<ll> st;
5     SegTree(const vector<ll> &v) : n((int)v.size()), st(n * 4 + 1, LLONG_MAX) {
6         for (int i = 0; i < n; ++i) update(i, v[i]);
7     }
8     void update(int p, ll v) { update(1, 0, n - 1, p, v); }
9     ll RMQ(int l, int r) { return RMQ(1, 0, n - 1, l, r); }
10
11 private:
12     void update(int node, int l, int r, int p, ll v) {
13         if (p < l or p > r) return; // fora do intervalo.
14
15         if (l == r) {
16             st[node] = v;
17             return;
18         }
19
20         int mid = l + (r - l) / 2;
21
22         update(node * 2, l, mid, p, v);
23         update(node * 2 + 1, mid + 1, r, p, v);
24
25         st[node] = min(st[node * 2], st[node * 2 + 1]);
26     }
27
28     ll RMQ(int node, int L, int R, int l, int r) {
29         if (l <= L and r >= R) return st[node];
30         if (L > r or R < l) return LLONG_MAX;
31         if (L == R) return st[node];
32
33         int mid = L + (R - L) / 2;
34
35         return min(RMQ(node * 2, L, mid, l, r),
36                   RMQ(node * 2 + 1, mid + 1, R, l, r));
37     }
38 };

```

## 1.5 Segtree Rsq Lazy Range Sum

```

1 struct SegTree {
2     int N;
3     vector<ll> ns, lazy;
4
5     SegTree(const vector<ll> &xs) : N(xs.size()), ns(4 * N, 0), lazy(4 * N, 0) {
6         for (size_t i = 0; i < xs.size(); ++i) update(i, i, xs[i]);
7     }
8
9     void update(int a, int b, ll value) { update(1, 0, N - 1, a, b, value); }
10
11     void update(int node, int L, int R, int a, int b, ll value) {

```

```

12 // Lazy propagation
13 if (lazy[node]) {
14     ns[node] += (R - L + 1) * lazy[node];
15
16     if (L < R) // Se o nó não é uma folha, propaga
17     {
18         lazy[2 * node] += lazy[node];
19         lazy[2 * node + 1] += lazy[node];
20     }
21
22     lazy[node] = 0;
23 }
24
25 if (a > R or b < L) return;
26
27 if (a <= L and R <= b) {
28     ns[node] += (R - L + 1) * value;
29
30     if (L < R) {
31         lazy[2 * node] += value;
32         lazy[2 * node + 1] += value;
33     }
34
35     return;
36 }
37
38 update(2 * node, L, (L + R) / 2, a, b, value);
39 update(2 * node + 1, (L + R) / 2 + 1, R, a, b, value);
40
41 ns[node] = ns[2 * node] + ns[2 * node + 1];
42 }
43
44 ll RSQ(int a, int b) { return RSQ(1, 0, N - 1, a, b); }
45
46 ll RSQ(int node, int L, int R, int a, int b) {
47     if (lazy[node]) {
48         ns[node] += (R - L + 1) * lazy[node];
49
50         if (L < R) {
51             lazy[2 * node] += lazy[node];
52             lazy[2 * node + 1] += lazy[node];
53         }
54
55         lazy[node] = 0;
56     }
57
58     if (a > R or b < L) return 0;
59
60     if (a <= L and R <= b) return ns[node];
61
62     ll x = RSQ(2 * node, L, (L + R) / 2, a, b);
63     ll y = RSQ(2 * node + 1, (L + R) / 2 + 1, R, a, b);
64
65     return x + y;
66 }
67 };

```

## 1.6 Segtree Rxq Lazy Range Xor

```
1 struct SegTree {
2     int N;
3     vector<ll> ns, lazy;
4
5     SegTree(const vector<ll> &xs) : N(xs.size()), ns(4 * N, 0), lazy(4 * N, 0) {
6         for (size_t i = 0; i < xs.size(); ++i) update(i, i, xs[i]);
7     }
8
9     void update(int a, int b, ll value) { update(1, 0, N - 1, a, b, value); }
10
11     void update(int node, int L, int R, int a, int b, ll value) {
12         // Lazy propagation
13         if (lazy[node]) {
14             ns[node] ^= lazy[node];
15
16             if (L < R) // Se o nó não é uma folha, propaga
17             {
18                 lazy[2 * node] ^= lazy[node];
19                 lazy[2 * node + 1] ^= lazy[node];
20             }
21
22             lazy[node] = 0;
23         }
24
25         if (a > R or b < L) return;
26
27         if (a <= L and R <= b) {
28             ns[node] ^= value;
29
30             if (L < R) {
31                 lazy[2 * node] ^= value;
32                 lazy[2 * node + 1] ^= value;
33             }
34
35             return;
36         }
37
38         update(2 * node, L, (L + R) / 2, a, b, value);
39         update(2 * node + 1, (L + R) / 2 + 1, R, a, b, value);
40
41         ns[node] = ns[2 * node] ^ ns[2 * node + 1];
42     }
43
44     ll rxq(int a, int b) { return RSQ(1, 0, N - 1, a, b); }
45
46     ll rxq(int node, int L, int R, int a, int b) {
47         if (lazy[node]) {
48             ns[node] ^= lazy[node];
49
50             if (L < R) {
51                 lazy[2 * node] ^= lazy[node];
52                 lazy[2 * node + 1] ^= lazy[node];
53             }
54
55             lazy[node] = 0;
56         }
```

```
57
58         if (a > R or b < L) return 0;
59
60         if (a <= L and R <= b) return ns[node];
61
62         ll x = rxq(2 * node, L, (L + R) / 2, a, b);
63         ll y = rxq(2 * node + 1, (L + R) / 2 + 1, R, a, b);
64
65         return x ^ y;
66     }
67 };
```

## 1.7 Dsu

```
1 class DSU:
2     def __init__(self, n):
3         self.n = n
4         self.p = [x for x in range(0, n + 1)]
5         self.size = [0 for i in range(0, n + 1)]
6
7     def find_set(self, x): # log n
8         if self.p[x] == x:
9             return x
10        else:
11            self.p[x] = self.find_set(self.p[x])
12            return self.p[x]
13
14    def same_set(self, x, y): # log n
15        return bool(self.find_set(x) == self.find_set(y))
16
17    def union_set(self, x, y): # log n
18        px = self.find_set(x)
19        py = self.find_set(y)
20
21        if px == py:
22            return
23
24        size_x = self.size[px]
25        size_y = self.size[py]
26
27        if size_x > size_y:
28            self.p[py] = self.p[px]
29            self.size[px] += self.size[py]
30        else:
31            self.p[px] = self.p[py]
32            self.size[py] += self.size[px]
```

## 1.8 Dsu

```
1 struct DSU {
2     vector<int> ps;
3     vector<int> size;
4     DSU(int N) : ps(N + 1), size(N + 1, 1) { iota(ps.begin(), ps.end(), 0); }
5     int find_set(int x) { return ps[x] == x ? x : ps[x] = find_set(ps[x]); }
6     bool same_set(int x, int y) { return find_set(x) == find_set(y); }
7     void union_set(int x, int y) {
8         if (same_set(x, y)) return;
9     }
```

```

10     int px = find_set(x);
11     int py = find_set(y);
12
13     if (size[px] < size[py]) swap(px, py);
14
15     ps[py] = px;
16     size[px] += size[py];
17 }
18 };

```

## 1.9 Sparse Table Rmq

```

1 /*
2     Sparse table implementation for rmq.
3     build: O(NlogN)
4     query: O(1)
5 */
6 int fastlog2(ll x) {
7     ull i = x;
8     return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
9 }
10 template <typename T>
11 class SparseTable {
12 public:
13     int N;
14     int K;
15     vector<vector<T>> st;
16     SparseTable(vector<T> vs)
17         : N((int)vs.size()), K(fastlog2(N) + 1), st(K + 1, vector<T>(N + 1)) {
18         copy(vs.begin(), vs.end(), st[0].begin());
19
20         for (int i = 1; i <= K; ++i)
21             for (int j = 0; j + (1 << i) <= N; ++j)
22                 st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
23     }
24     T RMQ(int l, int r) { // [l, r], 0 indexed
25         int i = fastlog2(r - l + 1);
26         return min(st[i][l], st[i][r - (1 << i) + 1]);
27     }
28 };

```

## 2 graphs

### 2.1 Scc-nodes-(kosajaru)

```

1 /*
2  * O(n+m)
3  * Returns a pair <a, b>
4  *     a: number of SCCs
5  *     b: vector of size n, where b[i] is the SCC id of node i
6  */
7 void dfs(ll u, vchar &visited, const vll2d &g, vll &scc, bool buildScc, ll id,
8         vll &sccid) {
9     visited[u] = true;
10    sccid[u] = id;
11    for (auto &v : g[u])
12        if (!visited[v]) dfs(v, visited, g, scc, buildScc, id, sccid);

```

```

13
14    // if it's the first pass, add the node to the scc
15    if (buildScc) scc.eb(u);
16 }
17
18 pair<ll, vll> kosajaru(vll2d &g) {
19     ll n = len(g);
20     vll scc;
21     vchar vis(n);
22     vll sccid(n);
23     for (ll i = 0; i < n; i++)
24         if (!vis[i]) dfs(i, vis, g, scc, true, 0, sccid);
25
26     // build the transposed graph
27     vll2d gt(n);
28     for (int i = 0; i < n; ++i)
29         for (auto &v : g[i]) gt[v].eb(i);
30
31     // run the dfs on the previous scc order
32     ll id = 1;
33     vis.assign(n, false);
34     for (ll i = len(scc) - 1; i >= 0; i--)
35         if (!vis[scc[i]]) {
36             dfs(scc[i], vis, gt, scc, false, id++, sccid);
37         }
38     return {id - 1, sccid};
39 }

```

### 2.2 2-sat-(struct)

```

1 struct SAT2 {
2     ll n;
3     vll2d adj, adj_t;
4     vc used;
5     vll order, comp;
6     vc assignment;
7     bool solvable;
8     SAT2(ll _n)
9         : n(2 * _n),
10           adj(n),
11           adj_t(n),
12           used(n),
13           order(n),
14           comp(n, -1),
15           assignment(n / 2) {}
16     void dfs1(int v) {
17         used[v] = true;
18         for (int u : adj[v]) {
19             if (!used[u]) dfs1(u);
20         }
21         order.push_back(v);
22     }
23
24     void dfs2(int v, int c1) {
25         comp[v] = c1;
26         for (int u : adj_t[v]) {
27             if (comp[u] == -1) dfs2(u, c1);
28         }

```

```

29 }
30
31 bool solve_2SAT() {
32     // find and label each SCC
33     for (int i = 0; i < n; ++i) {
34         if (!used[i]) dfs1(i);
35     }
36     reverse(all(order));
37     ll j = 0;
38     for (auto &v : order) {
39         if (comp[v] == -1) dfs2(v, j++);
40     }
41
42     assignment.assign(n / 2, false);
43     for (int i = 0; i < n; i += 2) {
44         // x and !x belong to the same SCC
45         if (comp[i] == comp[i + 1]) {
46             solvable = false;
47             return false;
48         }
49
50         assignment[i / 2] = comp[i] > comp[i + 1];
51     }
52     solvable = true;
53     return true;
54 }
55
56 void add_disjunction(int a, bool na, int b, bool nb) {
57     a = (2 * a) ^ na;
58     b = (2 * b) ^ nb;
59     int neg_a = a ^ 1;
60     int neg_b = b ^ 1;
61     adj[neg_a].push_back(b);
62     adj[neg_b].push_back(a);
63     adj_t[b].push_back(neg_a);
64     adj_t[a].push_back(neg_b);
65 }
66 };

```

## 2.3 Floyd Warshall

```

1 vector<vll> floyd_warshall(const vector<vll> &adj, ll n) {
2     auto dist = adj;
3
4     for (int i = 0; i < n; ++i) {
5         for (int j = 0; j < n; ++j) {
6             for (int k = 0; k < n; ++k) {
7                 dist[j][k] = min(dist[j][k], dist[j][i] + dist[i][k]);
8             }
9         }
10    }
11    return dist;
12 }

```

## 2.4 Topological-sorting

```

1 /*
2  * O(V)

```

```

3  * assumes:
4  *         * vertices have index [0, n-1]
5  * if is a DAG:
6  *         * returns a topological sorting
7  * else:
8  *         * returns an empty vector
9  */
10 enum class state { not_visited, processing, done };
11 bool dfs(const vector<vll> &adj, ll s, vector<state> &states, vll &order) {
12     states[s] = state::processing;
13     for (auto &v : adj[s]) {
14         if (states[v] == state::not_visited) {
15             if (not dfs(adj, v, states, order)) return false;
16         } else if (states[v] == state::processing)
17             return false;
18     }
19     states[s] = state::done;
20     order.pb(s);
21     return true;
22 }
23 vll topologicalSorting(const vector<vll> &adj) {
24     ll n = len(adj);
25     vll order;
26     vector<state> states(n, state::not_visited);
27     for (int i = 0; i < n; ++i) {
28         if (states[i] == state::not_visited) {
29             if (not dfs(adj, i, states, order)) return {};
30         }
31     }
32     reverse(all(order));
33     return order;
34 }

```

## 2.5 Lowest Common Ancestor Sparse Table

```

1 int fastlog2(ll x) {
2     ull i = x;
3     return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
4 }
5 template <typename T>
6 class SparseTable {
7 public:
8     int N;
9     int K;
10    vector<vector<T>> st;
11    SparseTable(vector<T> vs)
12        : N((int)vs.size()), K(fastlog2(N) + 1), st(K + 1, vector<T>(N + 1)) {
13        copy(vs.begin(), vs.end(), st[0].begin());
14
15        for (int i = 1; i <= K; ++i)
16            for (int j = 0; j + (1 << i) <= N; ++j)
17                st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
18    }
19    SparseTable() {}
20    T RMQ(int l, int r) {
21        int i = fastlog2(r - l + 1);
22        return min(st[i][l], st[i][r - (1 << i) + 1]);
23    }

```

```

24 };
25 class LCA {
26 public:
27     int p;
28     int n;
29     vi first;
30     vector<char> visited;
31     vi vertices;
32     vi height;
33     SparseTable<int> st;
34
35     LCA(const vector<vi> &g)
36         : p(0), n((int)g.size()), first(n + 1), visited(n + 1, 0), height(n + 1) {
37         build_dfs(g, 1, 1);
38         st = SparseTable<int>(vertices);
39     }
40
41     void build_dfs(const vector<vi> &g, int u, int hi) {
42         visited[u] = true;
43         height[u] = hi;
44         first[u] = vertices.size();
45         vertices.push_back(u);
46         for (auto uv : g[u]) {
47             if (!visited[uv]) {
48                 build_dfs(g, uv, hi + 1);
49                 vertices.push_back(uv);
50             }
51         }
52     }
53
54     int lca(int a, int b) {
55         int l = min(first[a], first[b]);
56         int r = max(first[a], first[b]);
57         return st.RMQ(l, r);
58     }
59 };

```

## 2.6 Count-scc-(kosajaru)

```

1 void dfs(ll u, vchar &visited, const vll2d &g, vll &scc, bool buildScc) {
2     visited[u] = true;
3     for (auto &v : g[u])
4         if (!visited[v]) dfs(v, visited, g, scc, buildScc);
5
6     // if it's the first pass, add the node to the scc
7     if (buildScc) scc.eb(u);
8 }
9
10 ll kosajaru(vll2d &g) {
11     ll n = len(g);
12     vll scc;
13     vchar vis(n);
14     for (ll i = 0; i < n; i++)
15         if (!vis[i]) dfs(i, vis, g, scc, true);
16
17     // build the transposed graph
18     vll2d gt(n);
19     for (int i = 0; i < n; ++i)

```

```

20         for (auto &v : g[i]) gt[v].eb(i);
21
22     // run the dfs on the previous scc order
23     ll scccnt = 0;
24     vis.assign(n, false);
25     for (ll i = len(scc) - 1; i >= 0; i--)
26         if (!vis[scc[i]]) dfs(scc[i], vis, gt, scc, false), scccnt++;
27     return scccnt;
28 }

```

## 2.7 Kruskal

```

1 class DSU:
2     def __init__(self, n):
3         self.n = n
4         self.p = [x for x in range(0, n + 1)]
5         self.size = [0 for i in range(0, n + 1)]
6
7     def find_set(self, x):
8         if self.p[x] == x:
9             return x
10        else:
11            self.p[x] = self.find_set(self.p[x])
12            return self.p[x]
13
14    def same_set(self, x, y):
15        return bool(self.find_set(x) == self.find_set(y))
16
17    def union_set(self, x, y):
18        px = self.find_set(x)
19        py = self.find_set(y)
20
21        if px == py:
22            return
23
24        size_x = self.size[px]
25        size_y = self.size[py]
26
27        if size_x > size_y:
28            self.p[py] = self.p[px]
29            self.size[px] += self.size[py]
30        else:
31            self.p[px] = self.p[py]
32            self.size[py] += self.size[px]
33
34    def kruskal(gv, n):
35        """
36        Receives te list of edges as a list of tuple in the form:
37            d, u, v
38            d: distance between u and v
39        And also n as the total of verties.
40        """
41        dsu = DSU(n)
42
43        c = 0
44        for e in gv:
45            d, u, v = e

```

```

47     if not dsu.same_set(u, v):
48         c += d
49         dsu.union_set(u, v)
50
51     return c

```

## 2.8 Scc(struct)

```

1 struct SCC {
2     ll N;
3     vll2d adj, tadj;
4     vll todo, comps, comp;
5     vector<set<ll>> sccadj;
6     vchar vis;
7     SCC(ll _N) : N(_N), adj(_N), tadj(_N), comp(_N, -1), sccadj(_N), vis(_N) {}
8
9     void add_edge(ll x, ll y) { adj[x].eb(y), tadj[y].eb(x); }
10
11     void dfs(ll x) {
12         vis[x] = 1;
13         for (auto &y : adj[x])
14             if (!vis[y]) dfs(y);
15         todo.pb(x);
16     }
17     void dfs2(ll x, ll v) {
18         comp[x] = v;
19         for (auto &y : tadj[x])
20             if (comp[y] == -1) dfs2(y, v);
21     }
22     void gen() {
23         for (ll i = 0; i < N; ++i)
24             if (!vis[i]) dfs(i);
25         reverse(all(todo));
26         for (auto &x : todo)
27             if (comp[x] == -1) {
28                 dfs2(x, x);
29                 comps.pb(x);
30             }
31     }
32
33     void genSCCGraph() {
34         for (ll i = 0; i < N; ++i) {
35             for (auto &j : adj[i]) {
36                 if (comp[i] != comp[j]) {
37                     sccadj[comp[i]].insert(comp[j]);
38                 }
39             }
40         }
41     }
42 };

```

## 2.9 Check-bipartite

```

1 // O(V)
2 bool checkBipartite(const ll n, const vector<vll> &adj) {
3     ll s = 0;
4     queue<ll> q;
5     q.push(s);

```

```

6     vll color(n, INF);
7     color[s] = 0;
8     bool isBipartite = true;
9     while (!q.empty() && isBipartite) {
10         ll u = q.front();
11         q.pop();
12         for (auto &v : adj[u]) {
13             if (color[v] == INF) {
14                 color[v] = 1 - color[u];
15                 q.push(v);
16             } else if (color[v] == color[u]) {
17                 return false;
18             }
19         }
20     }
21     return true;
22 }

```

## 2.10 Dijkstra

```

1 ll __inf = LLONG_MAX >> 5;
2 vll dijkstra(const vector<vector<pll>> &g, ll n) {
3     priority_queue<pll, vector<pll>, greater<pll>> pq;
4     vll dist(n, __inf);
5     vector<char> vis(n);
6     pq.emplace(0, 0);
7     dist[0] = 0;
8     while (!pq.empty()) {
9         auto [d1, v] = pq.top();
10        pq.pop();
11        if (vis[v]) continue;
12        vis[v] = true;
13
14        for (auto [d2, u] : g[v]) {
15            if (dist[u] > d1 + d2) {
16                dist[u] = d1 + d2;
17                pq.emplace(dist[u], u);
18            }
19        }
20    }
21    return dist;
22 }

```

## 3 extras

### 3.1 Binary To Gray

```

1 string binToGray(string bin) {
2     string gray(bin.size(), '0');
3     int n = bin.size() - 1;
4     gray[0] = bin[0];
5     for (int i = 1; i <= n; i++) {
6         gray[i] = '0' + (bin[i - 1] == '1') ^ (bin[i] == '1');
7     }
8     return gray;
9 }

```



## 3.2 Bigint

```
1 const int maxn = 1e2 + 14, lg = 15;
2 const int base = 1000000000;
3 const int base_digits = 9;
4 struct bigint {
5     vector<int> a;
6     int sign;
7
8     int size() {
9         if (a.empty()) return 0;
10        int ans = (a.size() - 1) * base_digits;
11        int ca = a.back();
12        while (ca) ans++, ca /= 10;
13        return ans;
14    }
15    bigint operator^(const bigint &v) {
16        bigint ans = 1, a = *this, b = v;
17        while (!b.isZero()) {
18            if (b % 2) ans *= a;
19            a *= a, b /= 2;
20        }
21        return ans;
22    }
23    string to_string() {
24        stringstream ss;
25        ss << *this;
26        string s;
27        ss >> s;
28        return s;
29    }
30    int sumof() {
31        string s = to_string();
32        int ans = 0;
33        for (auto c : s) ans += c - '0';
34        return ans;
35    }
36    /*</arpa>*/
37    bigint() : sign(1) {}
38
39    bigint(long long v) { *this = v; }
40
41    bigint(const string &s) { read(s); }
42
43    void operator=(const bigint &v) {
44        sign = v.sign;
45        a = v.a;
46    }
47
48    void operator=(long long v) {
49        sign = 1;
50        a.clear();
51        if (v < 0) sign = -1, v = -v;
52        for (; v > 0; v = v / base) a.push_back(v % base);
53    }
54
55    bigint operator+(const bigint &v) const {
56        if (sign == v.sign) {
```

```
57        bigint res = v;
58
59        for (int i = 0, carry = 0; i < (int)max(a.size(), v.a.size()) || carry;
60            ++i) {
61            if (i == (int)res.a.size()) res.a.push_back(0);
62            res.a[i] += carry + (i < (int)a.size() ? a[i] : 0);
63            carry = res.a[i] >= base;
64            if (carry) res.a[i] -= base;
65        }
66        return res;
67    }
68    return *this - (-v);
69 }
70
71 bigint operator-(const bigint &v) const {
72     if (sign == v.sign) {
73         if (abs() >= v.abs()) {
74             bigint res = *this;
75             for (int i = 0, carry = 0; i < (int)v.a.size() || carry; ++i) {
76                 res.a[i] -= carry + (i < (int)v.a.size() ? v.a[i] : 0);
77                 carry = res.a[i] < 0;
78                 if (carry) res.a[i] += base;
79             }
80             res.trim();
81             return res;
82         }
83         return -(v - *this);
84     }
85     return *this + (-v);
86 }
87
88 void operator*=(int v) {
89     if (v < 0) sign = -sign, v = -v;
90     for (int i = 0, carry = 0; i < (int)a.size() || carry; ++i) {
91         if (i == (int)a.size()) a.push_back(0);
92         long long cur = a[i] * (long long)v + carry;
93         carry = (int)(cur / base);
94         a[i] = (int)(cur % base);
95         // asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) :
96         // "A"(cur), "c"(base));
97     }
98     trim();
99 }
100
101 bigint operator*(int v) const {
102     bigint res = *this;
103     res *= v;
104     return res;
105 }
106
107 void operator*=(long long v) {
108     if (v < 0) sign = -sign, v = -v;
109     if (v > base) {
110         *this = *this * (v / base) * base + *this * (v % base);
111         return;
112     }
113     for (int i = 0, carry = 0; i < (int)a.size() || carry; ++i) {
114         if (i == (int)a.size()) a.push_back(0);
```

```

115     long long cur = a[i] * (long long)v + carry;
116     carry = (int)(cur / base);
117     a[i] = (int)(cur % base);
118     // asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) :
119     // "A"(cur), "c"(base));
120 }
121 trim();
122 }
123
124 bigint operator*(long long v) const {
125     bigint res = *this;
126     res *= v;
127     return res;
128 }
129
130 friend pair<bigint, bigint> divmod(const bigint &a1, const bigint &b1) {
131     int norm = base / (b1.a.back() + 1);
132     bigint a = a1.abs() * norm;
133     bigint b = b1.abs() * norm;
134     bigint q, r;
135     q.a.resize(a.a.size());
136
137     for (int i = a.a.size() - 1; i >= 0; i--) {
138         r *= base;
139         r += a.a[i];
140         int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
141         int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
142         int d = ((long long)base * s1 + s2) / b.a.back();
143         r -= b * d;
144         while (r < 0) r += b, --d;
145         q.a[i] = d;
146     }
147
148     q.sign = a1.sign * b1.sign;
149     r.sign = a1.sign;
150     q.trim();
151     r.trim();
152     return make_pair(q, r / norm);
153 }
154
155 bigint operator/(const bigint &v) const { return divmod(*this, v).first; }
156
157 bigint operator%(const bigint &v) const { return divmod(*this, v).second; }
158
159 void operator/=(int v) {
160     if (v < 0) sign = -sign, v = -v;
161     for (int i = (int)a.size() - 1; i >= 0; --i) {
162         long long cur = a[i] + rem * (long long)base;
163         a[i] = (int)(cur / v);
164         rem = (int)(cur % v);
165     }
166     trim();
167 }
168
169 bigint operator/(int v) const {
170     bigint res = *this;
171     res /= v;
172     return res;

```

```

173 }
174
175 int operator%(int v) const {
176     if (v < 0) v = -v;
177     int m = 0;
178     for (int i = a.size() - 1; i >= 0; --i)
179         m = (a[i] + m * (long long)base) % v;
180     return m * sign;
181 }
182
183 void operator+=(const bigint &v) { *this = *this + v; }
184 void operator-=(const bigint &v) { *this = *this - v; }
185 void operator*=(const bigint &v) { *this = *this * v; }
186 void operator/=(const bigint &v) { *this = *this / v; }
187
188 bool operator<(const bigint &v) const {
189     if (sign != v.sign) return sign < v.sign;
190     if (a.size() != v.a.size()) return a.size() * sign < v.a.size() * v.sign;
191     for (int i = a.size() - 1; i >= 0; i--)
192         if (a[i] != v.a[i]) return a[i] * sign < v.a[i] * sign;
193     return false;
194 }
195
196 bool operator>(const bigint &v) const { return v < *this; }
197 bool operator<=(const bigint &v) const { return !(v < *this); }
198 bool operator>=(const bigint &v) const { return !(*this < v); }
199 bool operator==(const bigint &v) const {
200     return !(*this < v) && !(v < *this);
201 }
202 bool operator!=(const bigint &v) const { return *this < v || v < *this; }
203
204 void trim() {
205     while (!a.empty() && !a.back()) a.pop_back();
206     if (a.empty()) sign = 1;
207 }
208
209 bool isZero() const { return a.empty() || (a.size() == 1 && !a[0]); }
210
211 bigint operator-() const {
212     bigint res = *this;
213     res.sign = -sign;
214     return res;
215 }
216
217 bigint abs() const {
218     bigint res = *this;
219     res.sign *= res.sign;
220     return res;
221 }
222
223 long long longValue() const {
224     long long res = 0;
225     for (int i = a.size() - 1; i >= 0; i--) res = res * base + a[i];
226     return res * sign;
227 }
228
229 friend bigint gcd(const bigint &a, const bigint &b) {
230     return b.isZero() ? a : gcd(b, a % b);

```

```

231 }
232 friend bigint lcm(const bigint &a, const bigint &b) {
233     return a / gcd(a, b) * b;
234 }
235
236 void read(const string &s) {
237     sign = 1;
238     a.clear();
239     int pos = 0;
240     while (pos < (int)s.size() && (s[pos] == '-' || s[pos] == '+')) {
241         if (s[pos] == '-') sign = -sign;
242         ++pos;
243     }
244     for (int i = s.size() - 1; i >= pos; i -= base_digits) {
245         int x = 0;
246         for (int j = max(pos, i - base_digits + 1); j <= i; j++)
247             x = x * 10 + s[j] - '0';
248         a.push_back(x);
249     }
250     trim();
251 }
252
253 friend istream &operator>>(istream &stream, bigint &v) {
254     string s;
255     stream >> s;
256     v.read(s);
257     return stream;
258 }
259
260 friend ostream &operator<<(ostream &stream, const bigint &v) {
261     if (v.sign == -1) stream << '-';
262     stream << (v.a.empty() ? 0 : v.a.back());
263     for (int i = (int)v.a.size() - 2; i >= 0; --i)
264         stream << setw(base_digits) << setfill('0') << v.a[i];
265     return stream;
266 }
267
268 static vector<int> convert_base(const vector<int> &a, int old_digits,
269                                int new_digits) {
270     vector<long long> p(max(old_digits, new_digits) + 1);
271     p[0] = 1;
272     for (int i = 1; i < (int)p.size(); i++) p[i] = p[i - 1] * 10;
273     vector<int> res;
274     long long cur = 0;
275     int cur_digits = 0;
276     for (int i = 0; i < (int)a.size(); i++) {
277         cur += a[i] * p[cur_digits];
278         cur_digits += old_digits;
279         while (cur_digits >= new_digits) {
280             res.push_back((int)(cur % p[new_digits]));
281             cur /= p[new_digits];
282             cur_digits -= new_digits;
283         }
284     }
285     res.push_back((int)cur);
286     while (!res.empty() && !res.back()) res.pop_back();
287     return res;
288 }

```

```

289
290 typedef vector<long long> vll;
291
292 static vll karatsubaMultiply(const vll &a, const vll &b) {
293     int n = a.size();
294     vll res(n + n);
295     if (n <= 32) {
296         for (int i = 0; i < n; i++)
297             for (int j = 0; j < n; j++) res[i + j] += a[i] * b[j];
298         return res;
299     }
300
301     int k = n >> 1;
302     vll a1(a.begin(), a.begin() + k);
303     vll a2(a.begin() + k, a.end());
304     vll b1(b.begin(), b.begin() + k);
305     vll b2(b.begin() + k, b.end());
306
307     vll a1b1 = karatsubaMultiply(a1, b1);
308     vll a2b2 = karatsubaMultiply(a2, b2);
309
310     for (int i = 0; i < k; i++) a2[i] += a1[i];
311     for (int i = 0; i < k; i++) b2[i] += b1[i];
312
313     vll r = karatsubaMultiply(a2, b2);
314     for (int i = 0; i < (int)a1b1.size(); i++) r[i] -= a1b1[i];
315     for (int i = 0; i < (int)a2b2.size(); i++) r[i] -= a2b2[i];
316
317     for (int i = 0; i < (int)r.size(); i++) res[i + k] += r[i];
318     for (int i = 0; i < (int)a1b1.size(); i++) res[i] += a1b1[i];
319     for (int i = 0; i < (int)a2b2.size(); i++) res[i + n] += a2b2[i];
320     return res;
321 }
322
323 bigint operator*(const bigint &v) const {
324     vector<int> a6 = convert_base(this->a, base_digits, 6);
325     vector<int> b6 = convert_base(v.a, base_digits, 6);
326     vll a(a6.begin(), a6.end());
327     vll b(b6.begin(), b6.end());
328     while (a.size() < b.size()) a.push_back(0);
329     while (b.size() < a.size()) b.push_back(0);
330     while (a.size() & (a.size() - 1)) a.push_back(0), b.push_back(0);
331     vll c = karatsubaMultiply(a, b);
332     bigint res;
333     res.sign = sign * v.sign;
334     for (int i = 0, carry = 0; i < (int)c.size(); i++) {
335         long long cur = c[i] + carry;
336         res.a.push_back((int)(cur % 1000000));
337         carry = (int)(cur / 1000000);
338     }
339     res.a = convert_base(res.a, 6, base_digits);
340     res.trim();
341     return res;
342 }
343 };

```

### 3.3 Get-permutation-cicles

```

1  /*
2  * receives a permutation [0, n-1]
3  * returns a vector of cycles
4  * for example: [ 1, 0, 3, 4, 2] -> [[0, 1], [2, 3, 4]]
5  * */
6  vector<vll> getPermutationCicles(const vll &ps) {
7      ll n = len(ps);
8      vector<char> visited(n);
9      vector<vll> cicles;
10     for (int i = 0; i < n; ++i) {
11         if (visited[i]) continue;
12
13         vll cicle;
14         ll pos = i;
15         while (!visited[pos]) {
16             cicle.pb(pos);
17             visited[pos] = true;
18             pos = ps[pos];
19         }
20
21         cicles.push_back(vll(all(cicle)));
22     }
23     return cicles;
24 }

```

## 4 dynamic programming

### 4.1 Edit Distance

```

1  int edit_distance(const string &a, const string &b) {
2      int n = a.size();
3      int m = b.size();
4      vector<vi> dp(n + 1, vi(m + 1, 0));
5
6      int ADD = 1, DEL = 1, CHG = 1;
7      for (int i = 0; i <= n; ++i) {
8          dp[i][0] = i * DEL;
9      }
10     for (int i = 1; i <= m; ++i) {
11         dp[0][i] = ADD * i;
12     }
13
14     for (int i = 1; i <= n; ++i) {
15         for (int j = 1; j <= m; ++j) {
16             int add = dp[i][j - 1] + ADD;
17             int del = dp[i - 1][j] + DEL;
18             int chg = dp[i - 1][j - 1] + (a[i - 1] == b[j - 1] ? 0 : 1) * CHG;
19             dp[i][j] = min({add, del, chg});
20         }
21     }
22
23     return dp[n][m];
24 }

```

### 4.2 Money Sum Bottom Up

```

1  /*

```

```

2      find every possible sum using
3      the given values only once.
4  */
5  set<int> money_sum(const vi &xs) {
6      using vc = vector<char>;
7      using vvc = vector<vc>;
8      int _m = accumulate(all(xs), 0);
9      int _n = xs.size();
10     vvc _dp(_n + 1, vc(_m + 1, 0));
11     set<int> _ans;
12     _dp[0][xs[0]] = 1;
13     for (int i = 1; i < _n; ++i) {
14         for (int j = 0; j <= _m; ++j) {
15             if (j == 0 or _dp[i - 1][j]) {
16                 _dp[i][j + xs[i]] = 1;
17                 _dp[i][j] = 1;
18             }
19         }
20     }
21
22     for (int i = 0; i < _n; ++i)
23         for (int j = 0; j <= _m; ++j)
24             if (_dp[i][j]) _ans.insert(j);
25     return _ans;
26 }

```

### 4.3 Knapsack Dp Values 01

```

1  const int MAX_N = 1001;
2  const int MAX_S = 100001;
3  array<array<int, MAX_S>, MAX_N> dp;
4  bool check[MAX_N][MAX_S];
5  pair<int, vi> knapsack(int S, const vector<pii> &xs) {
6      int N = (int)xs.size();
7
8      for (int i = 0; i <= N; ++i) dp[i][0] = 0;
9
10     for (int m = 0; m <= S; ++m) dp[0][m] = 0;
11
12     for (int i = 1; i <= N; ++i) {
13         for (int m = 1; m <= S; ++m) {
14             dp[i][m] = dp[i - 1][m];
15             check[i][m] = false;
16
17             auto [w, v] = xs[i - 1];
18
19             if (w <= m and (dp[i - 1][m - w] + v) >= dp[i][m]) {
20                 dp[i][m] = dp[i - 1][m - w] + v;
21                 check[i][m] = true;
22             }
23         }
24     }
25
26     int m = S;
27     vi es;
28
29     for (int i = N; i >= 1; --i) {
30         if (check[i][m]) {

```

```

31     es.push_back(i);
32     m -= xs[i - 1].first;
33 }
34 }
35
36 reverse(es.begin(), es.end());
37
38 return {dp[N][S], es};
39 }

```

## 4.4 Tsp

```

1 using vi = vector<int>;
2 vector<vi> dist;
3 vector<vi> memo;
4 /* 0 ( N^2 * 2^N )*/
5 int tsp(int i, int mask, int N) {
6     if (mask == (1 << N) - 1) return dist[i][0];
7     if (memo[i][mask] != -1) return memo[i][mask];
8     int ans = INT_MAX << 1;
9     for (int j = 0; j < N; ++j) {
10         if (mask & (1 << j)) continue;
11         auto t = tsp(j, mask | (1 << j), N) + dist[i][j];
12         ans = min(ans, t);
13     }
14     return memo[i][mask] = ans;
15 }

```

## 5 trees

### 5.1 Binary-lifting

```

1 /*
2  * far[h][i] = the node that 2^h far from node i
3  * sometimes is useful invert the order of loops
4  * time : O(nlogn)
5  */
6 const int maxlog = 20;
7 int far[maxlog + 1][n + 1];
8 int n;
9 for (int h = 1; h <= maxlog; h++) {
10     for (int i = 1; i <= n; i++) {
11         far[h][i] = far[h - 1][far[h - 1][i]];
12     }
13 }

```

### 5.2 Maximum-distances

```

1 /*
2  * Returns the maximum distance from every node to any other node in the tree.
3  */
4 pll mostDistantFrom(const vector<vll> &adj, ll n, ll root) {
5     // 0 indexed
6     ll mostDistantNode = root;
7     ll nodeDistance = 0;
8     queue<pll> q;

```

```

9     vector<char> vis(n);
10    q.emplace(root, 0);
11    vis[root] = true;
12    while (!q.empty()) {
13        auto [node, dist] = q.front();
14        q.pop();
15        if (dist > nodeDistance) {
16            nodeDistance = dist;
17            mostDistantNode = node;
18        }
19        for (auto u : adj[node]) {
20            if (!vis[u]) {
21                vis[u] = true;
22                q.emplace(u, dist + 1);
23            }
24        }
25    }
26    return {mostDistantNode, nodeDistance};
27 }
28
29 ll twoNodesDist(const vector<vll> &adj, ll n, ll a, ll b) {
30     queue<pll> q;
31     vector<char> vis(n);
32     q.emplace(a, 0);
33     while (!q.empty()) {
34         auto [node, dist] = q.front();
35         q.pop();
36         if (node == b) return dist;
37         for (auto u : adj[node]) {
38             if (!vis[u]) {
39                 vis[u] = true;
40                 q.emplace(u, dist + 1);
41             }
42         }
43     }
44     return -1;
45 }
46
47 tuple<ll, ll, ll> tree_diameter(const vector<vll> &adj, ll n) {
48     // returns two points of the diameter and the diameter itself
49     auto [node1, dist1] = mostDistantFrom(adj, n, 0);
50     auto [node2, dist2] = mostDistantFrom(adj, n, node1);
51     auto diameter = twoNodesDist(adj, n, node1, node2);
52     return make_tuple(node1, node2, diameter);
53 }
54
55 vll everyDistanceFromNode(const vector<vll> &adj, ll n, ll root) {
56     // Single Source Shortest Path, from a given root
57     queue<pair<ll, ll>> q;
58     vll ans(n, -1);
59     ans[root] = 0;
60     q.emplace(root, 0);
61     while (!q.empty()) {
62         auto [u, d] = q.front();
63         q.pop();
64
65         for (auto w : adj[u]) {
66             if (ans[w] != -1) continue;

```

```

67     ans[w] = d + 1;
68     q.emplace(w, d + 1);
69 }
70 }
71 return ans;
72 }
73
74 vll maxDistances(const vector<vll> &adj, ll n) {
75     auto [node1, node2, diameter] = tree_diameter(adj, n);
76     auto distances1 = everyDistanceFromNode(adj, n, node1);
77     auto distances2 = everyDistanceFromNode(adj, n, node2);
78     vll ans(n);
79     for (int i = 0; i < n; ++i) ans[i] = max(distances1[i], distances2[i]);
80     return ans;
81 }

```

## 5.3 Tree Diameter

```

1 pll mostDistantFrom(const vector<vll> &adj, ll n, ll root) {
2     // 0 indexed
3     ll mostDistantNode = root;
4     ll nodeDistance = 0;
5     queue<pll> q;
6     vector<char> vis(n);
7     q.emplace(root, 0);
8     vis[root] = true;
9     while (!q.empty()) {
10         auto [node, dist] = q.front();
11         q.pop();
12         if (dist > nodeDistance) {
13             nodeDistance = dist;
14             mostDistantNode = node;
15         }
16         for (auto u : adj[node]) {
17             if (!vis[u]) {
18                 vis[u] = true;
19                 q.emplace(u, dist + 1);
20             }
21         }
22     }
23     return {mostDistantNode, nodeDistance};
24 }
25 ll twoNodesDist(const vector<vll> &adj, ll n, ll a, ll b) {
26     // 0 indexed
27     queue<pll> q;
28     vector<char> vis(n);
29     q.emplace(a, 0);
30     while (!q.empty()) {
31         auto [node, dist] = q.front();
32         q.pop();
33         if (node == b) {
34             return dist;
35         }
36         for (auto u : adj[node]) {
37             if (!vis[u]) {
38                 vis[u] = true;
39                 q.emplace(u, dist + 1);
40             }

```

```

41     }
42 }
43 return -1;
44 }
45 ll tree_diameter(const vector<vll> &adj, ll n) {
46     // 0 indexed !!!
47     auto [node1, dist1] = mostDistantFrom(adj, n, 0);
48     auto [node2, dist2] = mostDistantFrom(adj, n, node1);
49     auto diameter = twoNodesDist(adj, n, node1, node2);
50     return diameter;
51 }

```

## 6 searching

### 6.1 Ternary Search Recursive

```

1 const double eps = 1e-6;
2
3 // IT MUST BE AN UNIMODAL FUNCTION
4 double f(int x) { return x * x + 2 * x + 4; }
5
6 double ternary_search(double l, double r) {
7     if (fabs(f(l) - f(r)) < eps) return f((l + (r - l) / 2.0));
8
9     auto third = (r - l) / 3.0;
10    auto m1 = l + third;
11    auto m2 = r - third;
12
13    // change the signal to find the maximum point.
14    return m1 < m2 ? ternary_search(m1, r) : ternary_search(l, m2);
15 }

```

## 7 math

### 7.1 Power-sum

```

1 // calculates K^0 + K^1 ... + K^n
2 ll fastpow(ll a, int n) {
3     if (n == 1) return a;
4     ll x = fastpow(a, n / 2);
5     return x * x * (n & 1 ? a : 1);
6 }
7 ll powersum(ll n, ll k) { return (fastpow(n, k + 1) - 1) / (n - 1); }

```

### 7.2 Sieve-list-primes

```

1 // lsit every prime until MAXN
2 const ll MAXN = 1e5;
3 vll list_primes(ll n) { // Nlog * log N
4     vll ps;
5     bitset<MAXN> sieve;
6     sieve.set();
7     sieve.reset(1);
8     for (ll i = 2; i <= n; ++i) {
9         if (sieve[i]) ps.push_back(i);

```

```

10     for (ll j = i * 2; j <= n; j += i) {
11         sieve.reset(j);
12     }
13 }
14 return ps;
15 }

```

## 7.3 Factorial

```

1 const ll MAX = 18;
2 vll fv(MAX, -1);
3 ll factorial(ll n) {
4     if (fv[n] != -1) return fv[n];
5     if (n == 0) return 1;
6     return n * factorial(n - 1);
7 }

```

## 7.4 Permutation-count

```

1 const ll MAX = 18;
2 vll fv(MAX, -1);
3 ll factorial(ll n) {
4     if (fv[n] != -1) return fv[n];
5     if (n == 0) return 1;
6     return n * factorial(n - 1);
7 }
8
9 template <typename T>
10 ll permutation_count(vector<T> xs) {
11     map<T, ll> h;
12     for (auto xi : xs) h[xi]++;
13     ll ans = factorial((ll)xs.size());
14     dbg(ans);
15     for (auto [v, cnt] : h) {
16         dbg(cnt);
17         ans /= cnt;
18     }
19
20     return ans;
21 }

```

## 7.5 N-choose-k-count

```

1 /*
2  * O(nm) time, O(m) space
3  * equal to n choose k
4  */
5 ll binom(ll n, ll k) {
6     if (k > n) return 0;
7     vll dp(k + 1, 0);
8     dp[0] = 1;
9     for (ll i = 1; i <= n; i++)
10         for (ll j = k; j > 0; j--) dp[j] = dp[j] + dp[j - 1];
11     return dp[k];
12 }

```

## 7.6 Gcd-using-factorization

```

1 // O(sqrt(n))
2 map<ll, ll> factorization(ll n) {
3     map<ll, ll> ans;
4     for (ll i = 2; i * i <= n; i++) {
5         ll count = 0;
6         for (; n % i == 0; count++, n /= i)
7             ;
8         if (count) ans[i] = count;
9     }
10    if (n > 1) ans[n]++;
11    return ans;
12 }
13
14 ll gcd_with_factorization(ll a, ll b) {
15     map<ll, ll> fa = factorization(a);
16     map<ll, ll> fb = factorization(b);
17     ll ans = 1;
18     for (auto fai : fa) {
19         ll k = min(fai.second, fb[fai.first]);
20         while (k--) ans *= fai.first;
21     }
22     return ans;
23 }

```

## 7.7 Is-prime

```

1 bool isprime(ll n) { // O(sqrt(n))
2     if (n < 2) return false;
3     if (n == 2) return true;
4     if (n % 2 == 0) return false;
5     for (ll i = 3; i * i < n; i += 2)
6         if (n % i == 0) return false;
7     return true;
8 }

```

## 7.8 Fast Exp

```

1 /*
2  Fast exponentiation algorithm,
3  compute a^n in O(log(n))
4  */
5 ll fexp(ll a, int n) {
6     if (n == 0) return 1;
7     if (n == 1) return a;
8     ll x = fexp(a, n / 2);
9     return x * x * (n & 1 ? a : 1);
10 }

```

## 7.9 Lcm-using-factorization

```

1 map<ll, ll> factorization(ll n) {
2     map<ll, ll> ans;
3     for (ll i = 2; i * i <= n; i++) {
4         ll count = 0;
5         for (; n % i == 0; count++, n /= i)
6             ;
7         if (count) ans[i] = count;

```

```

8   }
9   if (n > 1) ans[n]++;
10  return ans;
11 }
12
13 ll lcm_with_factorization(ll a, ll b) {
14     map<ll, ll> fa = factorization(a);
15     map<ll, ll> fb = factorization(b);
16     ll ans = 1;
17     for (auto fai : fa) {
18         ll k = max(fai.second, fb[fai.first]);
19         while (k--) ans *= fai.first;
20     }
21     return ans;
22 }

```

## 7.10 Euler-phi

```

1  const ll MAXN = 1e5;
2  vll list_primes(ll n) {    // Nlog * log N
3      vll ps;
4      bitset<MAXN> sieve;
5      sieve.set();
6      sieve.reset(1);
7      for (ll i = 2; i <= n; ++i) {
8          if (sieve[i]) ps.push_back(i);
9          for (ll j = i * 2; j <= n; j += i) {
10             sieve.reset(j);
11         }
12     }
13     return ps;
14 }
15
16 vector<pll> factorization(ll n, const vll &primes) {
17     vector<pll> ans;
18     for (auto &p : primes) {
19         if (n == 1) break;
20         ll cnt = 0;
21         while (n % p == 0) {
22             cnt++;
23             n /= p;
24         }
25         if (cnt) ans.emplace_back(p, cnt);
26     }
27     return ans;
28 }
29
30 ll phi(ll n, vector<pll> factors) {
31     if (n == 1) return 1;
32     ll ans = n;
33
34     for (auto [p, k] : factors) {
35         ans /= p;
36         ans *= (p - 1);
37     }
38
39     return ans;
40 }

```

## 7.11 Polynomial

```

1  using polynomial = vector<ll>;
2  int degree(const polynomial &xs) { return xs.size() - 1; }
3  ll horner_evaluate(const polynomial &xs, ll x) {
4      ll ans = 0;
5      ll n = degree(xs);
6      for (int i = n; i >= 0; --i) {
7          ans *= x;
8          ans += xs[i];
9      }
10     return ans;
11 }
12 polynomial operator+(const polynomial &a, const polynomial &b) {
13     int n = degree(a);
14     int m = degree(b);
15     polynomial r(max(n, m) + 1, 0);
16
17     for (int i = 0; i <= n; ++i) r[i] += a[i];
18     for (int j = 0; j <= m; ++j) r[j] += b[j];
19     while (!r.empty() and r.back() == 0) r.pop_back();
20     if (r.empty()) r.push_back(0);
21     return r;
22 }
23 polynomial operator*(const polynomial &p, const polynomial &q) {
24     int n = degree(p);
25     int m = degree(q);
26     polynomial r(n + m + 1, 0);
27     for (int i = 0; i <= n; ++i)
28         for (int j = 0; j <= m; ++j) r[i + j] += (p[i] * q[j]);
29     return r;
30 }

```

## 7.12 Integer Mod

```

1  const ll INF = 1e18;
2  const ll mod = 998244353;
3  template <ll MOD = mod>
4  struct Modular {
5      ll value;
6      static const ll MOD_value = MOD;
7
8      Modular(ll v = 0) {
9          value = v % MOD;
10         if (value < 0) value += MOD;
11     }
12     Modular(ll a, ll b) : value(0) {
13         *this += a;
14         *this /= b;
15     }
16
17     Modular& operator+=(Modular const& b) {
18         value += b.value;
19         if (value >= MOD) value -= MOD;
20         return *this;
21     }
22     Modular& operator-=(Modular const& b) {
23         value -= b.value;

```



```

24     if (value < 0) value += MOD;
25     return *this;
26 }
27 Modular& operator*=(Modular const& b) {
28     value = (ll)value * b.value % MOD;
29     return *this;
30 }
31
32 friend Modular mexp(Modular a, ll e) {
33     Modular res = 1;
34     while (e) {
35         if (e & 1) res *= a;
36         a *= a;
37         e >>= 1;
38     }
39     return res;
40 }
41 friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }
42
43 Modular& operator/=(Modular const& b) { return *this *= inverse(b); }
44 friend Modular operator+(Modular a, Modular const b) { return a += b; }
45 friend Modular operator++(int) { return this->value = (this->value + 1) % MOD; }
46 friend Modular operator++() { return this->value = (this->value + 1) % MOD; }
47 friend Modular operator-(Modular a, Modular const b) { return a -= b; }
48 friend Modular operator-(Modular const a) { return 0 - a; }
49 friend Modular operator--(int) {
50     return this->value = (this->value - 1 + MOD) % MOD;
51 }
52
53 friend Modular operator--() { return this->value = (this->value - 1 + MOD) % MOD; }
54 friend Modular operator*(Modular a, Modular const b) { return a *= b; }
55 friend Modular operator/(Modular a, Modular const b) { return a /= b; }
56 friend std::ostream& operator<<(std::ostream& os, Modular const& a) {
57     return os << a.value;
58 }
59 friend bool operator==(Modular const& a, Modular const& b) {
60     return a.value == b.value;
61 }
62 friend bool operator!=(Modular const& a, Modular const& b) {
63     return a.value != b.value;
64 }
65 };

```

## 7.13 Count Divisors Memo

```

1 const ll mod = 1073741824;
2 const ll maxd = 100 * 100 * 100 + 1;
3 vector<ll> memo(maxd, -1);
4 ll countdivisors(ll x) {
5     ll ox = x;
6     ll ans = 1;
7     for (ll i = 2; i <= x; ++i) {
8         if (memo[x] != -1) {
9             ans *= memo[x];
10            break;
11        }
12        ll count = 0;
13        while (x and x % i == 0) {

```

```

14            x /= i;
15            count++;
16        }
17        ans *= (count + 1);
18    }
19    memo[ox] = ans;
20    return ans;
21 }

```

## 7.14 Lcm

```

1 ll gcd(ll a, ll b) { return b ? gcd(b, a % b) : a; }
2 ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }

```

## 7.15 Factorial-factorization

```

1 // O(logN) greater k that p^k | n
2 ll E(ll n, ll p) {
3     ll k = 0, b = p;
4     while (b <= n) {
5         k += n / b;
6         b *= p;
7     }
8     return k;
9 }
10
11 // lsit every prime until MAXN O(Nlog * log N)
12 const ll MAXN = 1e5;
13 vll list_primes(ll n) {
14     vll ps;
15     bitset<MAXN> sieve;
16     sieve.set();
17     sieve.reset(1);
18     for (ll i = 2; i <= n; ++i) {
19         if (sieve[i]) ps.push_back(i);
20         for (ll j = i * 2; j <= n; j += i) sieve.reset(j);
21     }
22     return ps;
23 }
24
25 // O(pi(N)*logN)
26 map<ll, ll> factorial_factorization(ll n, const vll &primes) {
27     map<ll, ll> fs;
28     for (const auto &p : primes) {
29         if (p > n) break;
30         fs[p] = E(n, p);
31     }
32     return fs;
33 }

```

## 7.16 Factorization-with-primes

```

1 // Nlog * log N
2 const ll MAXN = 1e5;
3 vll list_primes(ll n) {
4     vll ps;
5     bitset<MAXN> sieve;

```

```

6   sieve.set();
7   sieve.reset(1);
8   for (ll i = 2; i <= n; ++i) {
9       if (sieve[i]) ps.push_back(i);
10      for (ll j = i * 2; j <= n; j += i) sieve.reset(j);
11  }
12  return ps;
13 }
14
15 // O(pi(sqrt(n)))
16 map<ll, ll> factorization(ll n, const vll &primes) {
17     map<ll, ll> ans;
18     for (auto p : primes) {
19         if (p * p > n) break;
20         ll count = 0;
21         for (; n % p == 0; count++, n /= p)
22             ;
23         if (count) ans[p] = count;
24     }
25     return ans;
26 }

```

## 7.17 Modular-inverse-using-phi

```

1 map<ll, ll> factorization(ll n) {
2     map<ll, ll> ans;
3     for (ll i = 2; i * i <= n; i++) {
4         ll count = 0;
5         for (; n % i == 0; count++, n /= i)
6             ;
7         if (count) ans[i] = count;
8     }
9     if (n > 1) ans[n]++;
10    return ans;
11 }
12
13 ll phi(ll n) {
14     if (n == 1) return 1;
15
16     auto fs = factorization(n);
17     auto res = n;
18
19     for (auto [p, k] : fs) {
20         res /= p;
21         res *= (p - 1);
22     }
23
24     return res;
25 }
26
27 ll fexp(ll a, ll n, ll mod) {
28     if (n == 0) return 1;
29     if (n == 1) return a;
30     ll x = fexp(a, n / 2, mod);
31     return x * x * (n & 1 ? a : 1) % mod;
32 }
33
34 ll inv(ll a, ll mod) { return fexp(a, phi(mod) - 1, mod); }

```

## 7.18 Factorization

```

1 // O(sqrt(n))
2 map<ll, ll> factorization(ll n) {
3     map<ll, ll> ans;
4     for (ll i = 2; i * i <= n; i++) {
5         ll count = 0;
6         for (; n % i == 0; count++, n /= i)
7             ;
8         if (count) ans[i] = count;
9     }
10    if (n > 1) ans[n]++;
11    return ans;
12 }

```

## 7.19 Gcd

```

1 ll gcd(ll a, ll b) { return b ? gcd(b, a % b) : a; }

```

## 7.20 Combinatorics With Repetitions

```

1 void combinations_with_repetition(int n, int k,
2                                   function<void(const vector<int> &)> process)
3 {
4     vector<int> v(k, 1);
5     int pos = k - 1;
6
7     while (true) {
8         process(v);
9
10        v[pos]++;
11
12        while (pos > 0 and v[pos] > n) {
13            --pos;
14            v[pos]++;
15        }
16
17        if (pos == 0 and v[pos] > n) break;
18
19        for (int i = pos + 1; i < k; ++i) v[i] = v[pos];
20
21        pos = k - 1;
22    }
23 }

```

# 8 strings

## 8.1 Rabin-karp

```

1 vi rabin_karp(string const &s, string const &t) {
2     ll p = 31;
3     ll m = 1e9 + 9;
4     int S = s.size(), T = t.size();
5
6     vll p_pow(max(S, T));
7     p_pow[0] = 1;

```

```

8   for (int i = 1; i < (int)p_pow.size(); i++) p_pow[i] = (p_pow[i - 1] * p) %
    m;
9
10  vll h(T + 1, 0);
11  for (int i = 0; i < T; i++)
12      h[i + 1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;
13  ll h_s = 0;
14  for (int i = 0; i < S; i++) h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;
15
16  vi occurrences;
17  for (int i = 0; i + S - 1 < T; i++) {
18      ll cur_h = (h[i + S] + m - h[i]) % m;
19      // IT DON'T CONSIDERE CONLIIONS !
20      if (cur_h == h_s * p_pow[i] % m) occurrences.push_back(i);
21  }
22  return occurrences;
23 }

```

## 8.2 Trie-naive

```

1  //   time: O(n^2) memory: O(n^2)
2  using Node = map<char, int>;
3  using vi = vector<int>;
4  using Trie = vector<Node>;
5
6  Trie build(const string &s) {
7      int n = (int)s.size();
8      Trie trie(1);
9      string suffix;
10
11     for (int i = n - 1; i >= 0; --i) {
12         suffix = s.substr(i) + '#';
13
14         int v = 0; // root
15         for (auto c : suffix) {
16             if (c == '#') { // makrs the poistion of an occurence
17                 trie[v][c] = i;
18                 break;
19             }
20             if (trie[v][c])
21                 v = trie[v][c];
22             else {
23                 trie.push_back({});
24                 trie[v][c] = trie.size() - 1;
25                 v = trie.size() - 1;
26             }
27         }
28     }
29     return trie;
30 }
31
32 vi search(Trie &trie, string s) {
33     int p = 0;
34     vi occ;
35     for (auto &c : s) {
36         p = trie[p][c];
37         if (!p) return occ;
38     }

```

```

39     queue<int> q;
40     q.push(0);
41     while (!q.empty()) {
42         auto cur = q.front();
43         q.pop();
44         for (auto [c, v] : trie[cur]) {
45             if (c == '#')
46                 occ.push_back(v);
47             else
48                 q.push(v);
49         }
50     }
51     return occ;
52 }
53
54 ll distinct_substr(const Trie &trie) {
55     ll cnt = 0;
56     queue<int> q;
57     q.push(0);
58     while (!q.empty()) {
59         auto u = q.front();
60         q.pop();
61
62         for (auto [c, v] : trie[u]) {
63             if (c != '#') {
64                 cnt++;
65                 q.push(v);
66             }
67         }
68     }
69     return cnt;
70 }
71 }

```

## 8.3 String-psum

```

1  struct strPsum {
2      ll n;
3      ll k;
4      vector<vll> psum;
5      strPsum(const string &s : n(s.size()), k(100), psum(k, vll(n + 1))) {
6          for (ll i = 1; i <= n; ++i) {
7              for (ll j = 0; j < k; ++j) {
8                  psum[j][i] = psum[j][i - 1];
9              }
10             psum[s[i - 1]][i]++;
11         }
12     }
13
14     ll qtd(ll l, ll r, char c) { // [0,n-1]
15         return psum[c][r + 1] - psum[c][l];
16     }
17 }

```