

<b>Contents</b>		
<b>1 Data structures</b>	<b>2</b>	
1.1 Bitree 2D	2	
1.2 Disjoint Sparse Table	2	
1.3 DSU/UFDS	2	
1.4 Ordered Set	3	
1.5 Prefix Sum 2D	3	
1.6 SegTree Range Sum Query Range PA sum/set Update	3	
1.7 SegTree Point Update (dynamic function)	4	
1.8 Segtree Range Max Query Point Max Assign Update (dynamic)	5	
1.9 Segtree Range Max Query Range Max Update	5	
1.10 SegTree Range Min Query Point Assign Update	6	
1.11 Segtree Range Sum Query Point Sum Update (dynamic)	6	
1.12 SegTree Range Xor Query Point Assign Update	7	
1.13 SegTree Range Min Query Range Sum Update	7	
1.14 SegTree Range Sum Query Range Sum Update	8	
1.15 Sparse Table	9	
<b>2 Dynamic programming</b>	<b>9</b>	
2.1 Binary Knapsack (bottom up)	9	
2.2 Binary Knapsack (top down)	9	
2.3 Edit Distance	10	
2.4 Kadane	10	
2.5 Longest Increasing Subsequence (LIS)	10	
2.6 Money Sum (Bottom Up)	10	
2.7 Travelling Salesman Problem	11	
<b>3 Geometry</b>	<b>11</b>	
3.1 Convex Hull	11	
3.2 Determinant	11	
3.3 Equals	11	
3.4 Line	11	
3.5 Point Struct And Utils (2d)	12	
3.6 Segment	12	
<b>4 Graphs</b>	<b>12</b>	
4.1 2 SAT	12	
4.2 Cycle Distances	13	
4.3 SCC (struct)	13	
4.4 Bellman-Ford (find negative cycle)	14	
4.5 Bellman Ford	14	
4.6 Binary Lifting	14	
4.7 Check Bipartitie	15	
4.8 Dijkstra (k Shortest Paths)	15	
4.9 Dijkstra (restore Path)	15	
4.10 Dijkstra	15	
4.11 Disjoint Edges Path (Maxflow)	16	
4.12 Euler Path (directed)	16	
4.13 Euler Path (undirected)	17	
4.14 Find Centroid	17	
4.15 Floyd Warshall	18	
4.16 Graph Cycle (directed)	18	
4.17 Graph Cycle (undirected)	18	
4.18 Kruskal	18	
4.19 Lowest Common Ancestor	19	
4.20 Tree Maximum Distance	19	
4.21 Maximum Flow (Edmonds-Karp)	20	
4.22 Minimum Cut (unweighted)	21	
4.23 Small to Large	21	
4.24 Sum every node distance	22	
4.25 Topological Sorting	22	
4.26 Tree Diameter	23	
<b>5 Math</b>	<b>23</b>	
5.1 GCD (with factorization)	23	
5.2 GCD	23	
5.3 LCM (with factorization)	23	
5.4 LCM	23	
5.5 Arithmetic Progression Sum	23	
5.6 Binomial MOD	24	
5.7 Binomial	24	
5.8 Euler phi $\varphi(n)$ (in range)	24	
5.9 Euler phi $\varphi(n)$	24	
5.10 Factorial Factorization	24	
5.11 Factorial	25	
5.12 Factorization (Pollard Rho)	25	
5.13 Factorization	25	
5.14 Fast Fourier Transform	25	
5.15 Fast pow	26	
5.16 Gauss Elimination	26	
5.17 Integer Mod	26	
5.18 Is prime	27	
5.19 Number of Divisors $\tau(n)$	27	
5.20 Power Sum	27	
5.21 Sieve list primes	27	
5.22 Sum of Divisors $\sigma(n)$	27	
<b>6 Problems</b>	<b>28</b>	
6.1 Hanoi Tower	28	
<b>7 Searching</b>	<b>28</b>	
7.1 Meet in the middle	28	
7.2 Ternary Search Recursive	28	
<b>8 Strings</b>	<b>28</b>	
8.1 Count Distinct Anagrams	28	
8.2 Double Hash Range Query	28	
8.3 Hash Range Query	29	
8.4 K-th digit in digit string	29	
8.5 Longest Palindrome Substring (Manacher)	29	
8.6 Rabin Karp	30	
8.7 String Psum	30	
8.8 Suffix Automaton (complete)	30	
8.9 Z-function get occurence positions	32	
<b>9 Settings and macros</b>	<b>32</b>	
9.1 short-macro.cpp	32	
9.2 .vimrc	32	
9.3 degug.cpp	32	
9.4 .bashrc	33	
9.5 macro.cpp	33	

# 1 Data structures

## 1.1 Bitree 2D

Given a 2d array allow you to sum  $val$  to the position  $(x, y)$  and find the sum of the rectangle with left top corner  $(x1, y1)$  and right bottom corner  $(x2, y2)$

Update and query 1 indexed !

Time: update  $O(\log n^2)$ , query  $O(\log n^2)$

```
struct Bit2d {
    int n;
    vll2d bit;
    Bit2d(int ni) : n(ni), bit(n + 1, vll(n + 1)) {}
    Bit2d(int ni, vll2d &xs) : n(ni), bit(n + 1, vll(n + 1)) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                update(i, j, xs[i][j]);
            }
        }
    }
    void update(int x, int y, ll val) {
        for (; x <= n; x += (x & (-x))) {
            for (int i = y; i <= n; i += (i & (-i))) {
                bit[x][i] += val;
            }
        }
    }
    ll sum(int x, int y) {
        ll ans = 0;

        for (int i = x; i; i -= (i & (-i))) {
            for (int j = y; j; j -= (j & (-j))) {
                ans += bit[i][j];
            }
        }
        return ans;
    }
    ll query(int x1, int y1, int x2, int y2) {
        return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) +
            sum(x1 - 1, y1 - 1);
    }
};
```

## 1.2 Disjoint Sparse Table

Answers queries of any monoid operation (i.e. has identity element and is associative)

Build:  $O(N \log N)$ , Query:  $O(1)$

```
#define F(expr) [](auto a, auto b) { return expr; }
template <typename T>
struct DisjointSparseTable {
    using Operation = T (*)(T, T);

    vector<vector<T>> st;
    Operation f;
    T identity;
```

```
static constexpr int log2_floor(unsigned long long i) noexcept {
    return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
}
```

// Lazy loading constructor. Needs to call build!

```
DisjointSparseTable(Operation op, const T neutral = T())
    : st(), f(op), identity(neutral) {}
```

```
DisjointSparseTable(vector<T> v) : DisjointSparseTable(v, F(min(a, b))) {}
```

```
DisjointSparseTable(vector<T> v, Operation op, const T neutral = T())
    : st(), f(op), identity(neutral) {
    build(v);
}
```

```
void build(vector<T> v) {
    st.resize(log2_floor(v.size()) + 1,
        vector<T>(1ll << (log2_floor(v.size()) + 1)));
    v.resize(st[0].size(), identity);
    for (int level = 0; level < (int)st.size(); ++level) {
        for (int block = 0; block < (1 << level); ++block) {
            const auto l = block << (st.size() - level);
            const auto r = (block + 1) << (st.size() - level);
            const auto m = l + (r - l) / 2;
```

```
            st[level][m] = v[m];
            for (int i = m + 1; i < r; i++)
                st[level][i] = f(st[level][i - 1], v[i]);
            st[level][m - 1] = v[m - 1];
            for (int i = m - 2; i >= l; i--)
                st[level][i] = f(st[level][i + 1], v[i]);
        }
    }
```

```
T query(int l, int r) const {
    if (l > r) return identity;
    if (l == r) return st.back()[l];
```

```
    const auto k = log2_floor(l ^ r);
    const auto level = (int)st.size() - 1 - k;
    return f(st[level][l], st[level][r]);
}
```

```
};
```

## 1.3 DSU/UFDS

Uncomment the lines to reover which element belong to each set.

Time:  $\approx O(1)$  for everything.

```
struct DSU {
    vi ps;
    vi size;
    // vector<unordered_set<int>> sts;
    DSU(int N) : ps(N + 1), size(N, 1), sts(N) {
        iota(all(ps), 0);
        // for (int i = 0; i < N; i++) sts[i].insert(i);
    }
```

```

int find_set(int x) { return ps[x] == x ? x : ps[x] = find_set(ps[x]); }
bool same_set(int x, int y) { return find_set(x) == find_set(y); }
void union_set(int x, int y) {
    if (same_set(x, y)) return;

    int px = find_set(x);
    int py = find_set(y);

    if (size[px] < size[py]) swap(px, py);

    ps[py] = px;
    size[px] += size[py];
    // sts[px].merge(sts[py]);
}
};

```

## 1.4 Ordered Set

If you need an ordered **multiset** you may add an id to each value. Using `greater_equal`, or `less_equal` is considered undefined behavior.

- **order\_of\_key(k)** : Number of items strictly smaller/greater than k .
- **find\_by\_order(k)** : K-th element in a set (counting from zero).

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

```

```

template <typename T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;

```

## 1.5 Prefix Sum 2D

Given an 2d array with  $n$  lines and  $m$  columns, find the sum of the subarray that have the left upper corner at  $(x1, y1)$  and right bottom corner at  $(x2, y2)$ .

Time: build  $O(n \cdot m)$ , query  $O(1)$ .

```

struct psum2d {
    vll2d s;
    vll2d psum;
    psum2d(vll2d &grid, int n, int m)
        : s(n + 1, vll(m + 1)), psum(n + 1, vll(m + 1)) {
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++) s[i][j] = s[i][j - 1] + grid[i][j];

        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= m; j++) psum[i][j] = psum[i - 1][j] + s[i][j];
    }

    ll query(int x1, int y1, int x2, int y2) {
        ll ans = psum[x2][y2] + psum[x1 - 1][y1 - 1];
        ans -= psum[x2][y1 - 1] + psum[x1 - 1][y2];
        return ans;
    }
};

```

## 1.6 SegTree Range Sum Query Range PA sum/set Update

Makes arithmetic progression updates in range and sum queries.

Considering  $PA(A, R) = [A + R, A + 2R, A + 3R, \dots]$

- **update\_set(l, r, A, R)**: sets  $[l, r]$  to  $PA(A, R)$
- **update\_add(l, r, A, R)**: sum  $PA(A, R)$  in  $[l, r]$
- **query(l, r)**: sum in range  $[l, r]$

**0 indexed !**

Time: build  $O(n)$ , updates and queries  $O(\log n)$

```

const ll oo = 1e18;
struct SegTree {
    struct Data {
        ll sum;
        ll set_a, set_r, add_a, add_r;
        Data() : sum(0), set_a(oo), set_r(0), add_a(0), add_r(0) {}
    };
    int n;
    vector<Data> seg;
    SegTree(int n_) : n(n_), seg(vector<Data>(4 * n)) {}

    void prop(int p, int l, int r) {
        int sz = r - l + 1;
        ll &sum = seg[p].sum, &set_a = seg[p].set_a, &set_r = seg[p].set_r,
            &add_a = seg[p].add_a, &add_r = seg[p].add_r;

        if (set_a != oo) {
            set_a += add_a, set_r += add_r;
            sum = set_a * sz + set_r * sz * (sz + 1) / 2;
            if (l != r) {
                int m = (l + r) / 2;

                seg[2 * p].set_a = set_a;
                seg[2 * p].set_r = set_r;
                seg[2 * p].add_a = seg[2 * p].add_r = 0;

                seg[2 * p + 1].set_a = set_a + set_r * (m - l + 1);
                seg[2 * p + 1].set_r = set_r;
                seg[2 * p + 1].add_a = seg[2 * p + 1].add_r = 0;
            }
            set_a = oo, set_r = 0;
            add_a = add_r = 0;
        } else if (add_a or add_r) {
            sum += add_a * sz + add_r * sz * (sz + 1) / 2;
            if (l != r) {
                int m = (l + r) / 2;

                seg[2 * p].add_a += add_a;
                seg[2 * p].add_r += add_r;

                seg[2 * p + 1].add_a += add_a + add_r * (m - l + 1);
                seg[2 * p + 1].add_r += add_r;
            }
            add_a = add_r = 0;
        }
    }
};

```

```

int inter(pii a, pii b) {
    if (a.first > b.first) swap(a, b);
    return max(0, min(a.second, b.second) - b.first + 1);
}

ll set(int a, int b, ll aa, ll rr, int p, int l, int r) {
    prop(p, l, r);
    if (b < l or r < a) return seg[p].sum;
    if (a <= l and r <= b) {
        seg[p].set_a = aa;
        seg[p].set_r = rr;
        prop(p, l, r);
        return seg[p].sum;
    }
    int m = (l + r) / 2;
    int tam_l = inter({l, m}, {a, b});
    return seg[p].sum = set(a, b, aa, rr, 2 * p, l, m) +
        set(a, b, aa + rr * tam_l, rr, 2 * p + 1, m + 1, r);
}

void update_set(int l, int r, ll aa, ll rr) {
    set(l, r, aa, rr, 1, 0, n - 1);
}

ll add(int a, int b, ll aa, ll rr, int p, int l, int r) {
    prop(p, l, r);
    if (b < l or r < a) return seg[p].sum;
    if (a <= l and r <= b) {
        seg[p].add_a += aa;
        seg[p].add_r += rr;
        prop(p, l, r);
        return seg[p].sum;
    }
    int m = (l + r) / 2;
    int tam_l = inter({l, m}, {a, b});
    return seg[p].sum = add(a, b, aa, rr, 2 * p, l, m) +
        add(a, b, aa + rr * tam_l, rr, 2 * p + 1, m + 1, r);
}

void update_add(int l, int r, ll aa, ll rr) {
    add(l, r, aa, rr, 1, 0, n - 1);
}

ll query(int a, int b, int p, int l, int r) {
    prop(p, l, r);
    if (b < l or r < a) return 0;
    if (a <= l and r <= b) return seg[p].sum;
    int m = (l + r) / 2;
    return query(a, b, 2 * p, l, m) + query(a, b, 2 * p + 1, m + 1, r);
}

ll query(int l, int r) { return query(l, r, 1, 0, n - 1); }
};

void run() {
    int n, q;
    cin >> n >> q;

    SegTree st(n);
    for (int i = 0; i < n; i++) {
        ll x;
        cin >> x;
        st.update_set(i, i, x, 0);
    }
}

```

```

while (q--) {
    int o;
    cin >> o;

    int a, b;
    cin >> a >> b;
    a--, b--;
    if (o == 1) {
        st.update_add(a, b, 0, 1);
    } else {
        cout << st.query(a, b) << endl;
    }
}
}

```

## 1.7 SegTree Point Update (dynamic function)

Answers queries of any monoid operation (i.e. has identity element and is associative)  
 Build:  $O(N)$ , Query:  $O(\log N)$

```

#define F(expr) [](auto a, auto b) { return expr; }
template <typename T>
struct SegTree {
    using Operation = T (*)(T, T);

    int N;
    vector<T> ns;
    Operation operation;
    T identity;

    SegTree(int n, Operation op = F(a + b), T neutral = T())
        : N(n), ns(2 * N, neutral), operation(op), identity(neutral) {}

    SegTree(const vector<T> &v, Operation op = F(a + b), T neutral = T())
        : SegTree((int)v.size(), op, neutral) {
            copy(v.begin(), v.end(), ns.begin() + N);

            for (int i = N - 1; i > 0; --i) ns[i] = operation(ns[2 * i], ns[2 * i + 1]);
        }

    T query(size_t i) const { return ns[i + N]; }

    T query(size_t l, size_t r) const {
        auto a = l + N, b = r + N;
        auto ans = identity;
        // Non-associative operations needs to be processed backwards
        stack<T> st;
        while (a <= b) {
            if (a & 1) ans = operation(ans, ns[a++]);
            if (not(b & 1)) st.push(ns[b--]);

            a >>= 1;
            b >>= 1;
        }

        for (; !st.empty(); st.pop()) ans = operation(ans, st.top());
    }
}

```

```

    return ans;
}

void update(size_t i, T value) { update_set(i, operation(ns[i + N], value));
}

void update_set(size_t i, T value) {
    auto a = i + N;

    ns[a] = value;
    while (a >= 1) ns[a] = operation(ns[2 * a], ns[2 * a + 1]);
}
};

```

## 1.8 Segtree Range Max Query Point Max Assign Update (dynamic)

Answers range queries in ranges until  $10^9$  (maybe more)  
 Time: query and update  $O(n \cdot \log n)$

```

struct node;
node *newNode();

struct node {
    node *left, *right;
    int lv, rv;
    ll val;

    node() : left(NULL), right(NULL), val(-oo) {}

    inline void init(int l, int r) {
        lv = l;
        rv = r;
    }

    inline void extend() {
        if (!left) {
            int m = (lv + rv) / 2;
            left = newNode();
            right = newNode();
            left->init(lv, m);
            right->init(m + 1, rv);
        }
    }

    ll query(int l, int r) {
        if (r < lv || rv < l) {
            return 0;
        }

        if (l <= lv && rv <= r) {
            return val;
        }

        extend();
        return max(left->query(l, r), right->query(l, r));
    }
}

```

```

void update(int p, ll newVal) {
    if (lv == rv) {
        val = max(val, newVal);
        return;
    }

    extend();
    (p <= left->rv ? left : right)->update(p, newVal);
    val = max(left->val, right->val);
}

};

const int BUFFSZ(1e7);
node *newNode() {
    static int bufSize = BUFFSZ;
    static node buf[(int)BUFFSZ];
    assert(bufSize);
    return &buf[--bufSize];
}

struct SegTree {
    int n;
    node *root;
    SegTree(int _n) : n(_n) {
        root = newNode();
        root->init(0, n);
    }

    ll query(int l, int r) { return root->query(l, r); }
    void update(int p, ll v) { root->update(p, v); }
};

```

## 1.9 Segtree Range Max Query Range Max Update

```

template <typename T = ll>
struct SegTree {
    int N;
    T nu, nq;
    vector<T> st, lazy;
    SegTree(const vector<T> &xs)
        : N(len(xs)),
          nu(numeric_limits<T>::min()),
          nq(numeric_limits<T>::min()),
          st(4 * N + 1, nu),
          lazy(4 * N + 1, nu) {
        for (int i = 0; i < len(xs); ++i) update(i, i, xs[i]);
    }

    void update(int l, int r, T value) { update(1, 0, N - 1, l, r, value); }

    T query(int l, int r) { return query(1, 0, N - 1, l, r); }

    void update(int node, int nl, int nr, int ql, int qr, T v) {
        propagation(node, nl, nr);

        if (ql > nr or qr < nl) return;
    }
}

```

```

    st[node] = max(st[node], v);
    if (ql <= nl and nr <= qr) {
        if (nl < nr) {
            lazy[left(node)] = max(lazy[left(node)], v);
            lazy[right(node)] = max(lazy[right(node)], v);
        }
        return;
    }
    update(left(node), nl, mid(nl, nr), ql, qr, v);
    update(right(node), mid(nl, nr) + 1, nr, ql, qr, v);

    st[node] = max(st[left(node)], st[right(node)]);
}

T query(int node, int nl, int nr, int ql, int qr) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return nq;

    if (ql <= nl and nr <= qr) return st[node];

    T x = query(left(node), nl, mid(nl, nr), ql, qr);
    T y = query(right(node), mid(nl, nr) + 1, nr, ql, qr);

    return max(x, y);
}

void propagation(int node, int nl, int nr) {
    if (lazy[node] != nu) {
        st[node] = max(st[node], lazy[node]);

        if (nl < nr) {
            lazy[left(node)] = max(lazy[left(node)], lazy[node]);
            lazy[right(node)] = max(lazy[right(node)], lazy[node]);
        }

        lazy[node] = nu;
    }
}

int left(int p) { return p << 1; }
int right(int p) { return (p << 1) + 1; }
int mid(int l, int r) { return (r - l) / 2 + 1; }
};

int main() {
    int n;
    cin >> n;
    vector<array<int, 3>> xs(n);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < 3; ++j) {
            cin >> xs[i][j];
        }
    }
    vi aux(n, 0);
    SegTree<int> st(aux);
    for (int i = 0; i < n; ++i) {
        int a = min(i + xs[i][1], n);
        int b = min(i + xs[i][2], n);

```

```

        st.update(i, i, st.query(i, i) + xs[i][0]);
        int cur = st.query(i, i);
        st.update(a, b, cur);
    }

    cout << st.query(0, n) << '\n';
}

```

## 1.10 SegTree Range Min Query Point Assign Update

```

template <typename T = ll>
struct SegTree {
    int n;
    T nu, nq;
    vector<T> st;
    SegTree(const vector<T> &v)
        : n(len(v)), nu(0), nq(numeric_limits<T>::max()), st(n * 4 + 1, nu) {
        for (int i = 0; i < n; ++i) update(i, v[i]);
    }
    void update(int p, T v) { update(1, 0, n - 1, p, v); }
    T query(int l, int r) { return query(1, 0, n - 1, l, r); }

    void update(int node, int nl, int nr, int p, T v) {
        if (p < nl or p > nr) return;

        if (nl == nr) {
            st[node] = v;
            return;
        }

        update(left(node), nl, mid(nl, nr), p, v);
        update(right(node), mid(nl, nr) + 1, nr, p, v);

        st[node] = min(st[left(node)], st[right(node)]);
    }

    T query(int node, int nl, int nr, int ql, int qr) {
        if (ql <= nl and qr >= nr) return st[node];
        if (nl > qr or nr < ql) return nq;
        if (nl == nr) return st[node];

        return min(query(left(node), nl, mid(nl, nr), ql, qr),
                    query(right(node), mid(nl, nr) + 1, nr, ql, qr));
    }

    int left(int p) { return p << 1; }
    int right(int p) { return (p << 1) + 1; }
    int mid(int l, int r) { return (r - l) / 2 + 1; }
};

```

## 1.11 Segtree Range Sum Query Point Sum Update (dynamic)

Answers range queries in ranges until  $10^9$  (maybe more)

Time: query and update  $O(n \cdot \log n)$

```

struct node;
node *newNode();

struct node {

```

```

node *left, *right;
int lv, rv;
ll val;

node() : left(NULL), right(NULL), val(0) {}

inline void init(int l, int r) {
    lv = l;
    rv = r;
}

inline void extend() {
    if (!left) {
        int m = (rv - lv) / 2 + lv;
        left = newNode();
        right = newNode();
        left->init(lv, m);
        right->init(m + 1, rv);
    }
}

ll query(int l, int r) {
    if (r < lv || rv < l) {
        return 0;
    }

    if (l <= lv && rv <= r) {
        return val;
    }

    extend();
    return left->query(l, r) + right->query(l, r);
}

void update(int p, ll newVal) {
    if (lv == rv) {
        val += newVal;
        return;
    }

    extend();
    (p <= left->rv ? left : right)->update(p, newVal);
    val = left->val + right->val;
}

};

const int BUFFSZ(1.3e7);
node *newNode() {
    static int bufSize = BUFFSZ;
    static node buf[(int)BUFFSZ];
    // assert(bufSize);
    return &buf[--bufSize];
}

struct SegTree {
    int n;
    node *root;
    SegTree(int _n) : n(_n) {

```

```

        root = newNode();
        root->init(0, n);
    }
    ll query(int l, int r) { return root->query(l, r); }
    void update(int p, ll v) { root->update(p, v); }
};

```

## 1.12 SegTree Range Xor Query Point Assign Update

```

template <typename T = ll>
struct SegTree {
    int n;
    T nu, nq;
    vector<T> st;
    SegTree(const vector<T> &v) : n(len(v)), nu(0), nq(0), st(n * 4 + 1, nu) {
        for (int i = 0; i < n; ++i) update(i, v[i]);
    }
    void update(int p, T v) { update(1, 0, n - 1, p, v); }
    T query(int l, int r) { return query(1, 0, n - 1, l, r); }

    void update(int node, int nl, int nr, int p, T v) {
        if (p < nl || p > nr) return;

        if (nl == nr) {
            st[node] = v;
            return;
        }

        update(left(node), nl, mid(nl, nr), p, v);
        update(right(node), mid(nl, nr) + 1, nr, p, v);

        st[node] = st[left(node)] ^ st[right(node)];
    }

    T query(int node, int nl, int nr, int ql, int qr) {
        if (ql <= nl && qr >= nr) return st[node];
        if (nl > qr || nr < ql) return nq;
        if (nl == nr) return st[node];

        return query(left(node), nl, mid(nl, nr), ql, qr) ^
            query(right(node), mid(nl, nr) + 1, nr, ql, qr);
    }

    int left(int p) { return p << 1; }
    int right(int p) { return (p << 1) + 1; }
    int mid(int l, int r) { return (r - l) / 2 + l; }
};

```

## 1.13 SegTree Range Min Query Range Sum Update

```

template <typename t = ll>
struct SegTree {
    int n;
    t nu;
    t nq;
    vector<t> st, lazy;

```

```

SegTree(const vector<t> &xs)
: n(len(xs)),
  nu(0),
  nq(numeric_limits<t>::max()),
  st(4 * n, nu),
  lazy(4 * n, nu) {
  for (int i = 0; i < len(xs); ++i) update(i, i, xs[i]);
}

SegTree(int n) : n(n), st(4 * n, nu), lazy(4 * n, nu) {}

void update(int l, int r, ll value) { update(1, 0, n - 1, l, r, value); }

t query(int l, int r) { return query(1, 0, n - 1, l, r); }

void update(int node, int nl, int nr, int ql, int qr, ll v) {
  propagation(node, nl, nr);

  if (ql > nr or qr < nl) return;

  if (ql <= nl and nr <= qr) {
    st[node] += (nr - nl + 1) * v;

    if (nl < nr) {
      lazy[left(node)] += v;
      lazy[right(node)] += v;
    }

    return;
  }

  update(left(node), nl, mid(nl, nr), ql, qr, v);
  update(right(node), mid(nl, nr) + 1, nr, ql, qr, v);

  st[node] = min(st[left(node)], st[right(node)]);
}

t query(int node, int nl, int nr, int ql, int qr) {
  propagation(node, nl, nr);

  if (ql > nr or qr < nl) return nq;

  if (ql <= nl and nr <= qr) return st[node];

  t x = query(left(node), nl, mid(nl, nr), ql, qr);
  t y = query(right(node), mid(nl, nr) + 1, nr, ql, qr);

  return min(x, y);
}

void propagation(int node, int nl, int nr) {
  if (lazy[node]) {
    st[node] += lazy[node];

    if (nl < nr) {
      lazy[left(node)] += lazy[node];
      lazy[right(node)] += lazy[node];
    }
  }
}

```

```

    lazy[node] = nu;
  }
}

int left(int p) { return p << 1; }
int right(int p) { return (p << 1) + 1; }
int mid(int l, int r) { return (r - l) / 2 + 1; }
};

```

## 1.14 SegTree Range Sum Query Range Sum Update

```

template <typename T = ll>
struct SegTree {
  int N;
  T nu;
  T nq;
  vector<T> st, lazy;
  SegTree(const vector<T> &xs)
    : N(len(xs)), nu(0), nq(0), st(4 * N, nu), lazy(4 * N, nu) {
    for (int i = 0; i < len(xs); ++i) update(i, i, xs[i]);
  }

  SegTree(int n) : N(n), nu(0), nq(0), st(4 * N, nu), lazy(4 * N, nu) {}

  void update(int l, int r, ll value) { update(1, 0, N - 1, l, r, value); }

  T query(int l, int r) { return query(1, 0, N - 1, l, r); }

  void update(int node, int nl, int nr, int ql, int qr, ll v) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return;

    if (ql <= nl and nr <= qr) {
      st[node] += (nr - nl + 1) * v;

      if (nl < nr) {
        lazy[left(node)] += v;
        lazy[right(node)] += v;
      }

      return;
    }

    update(left(node), nl, mid(nl, nr), ql, qr, v);
    update(right(node), mid(nl, nr) + 1, nr, ql, qr, v);

    st[node] = st[left(node)] + st[right(node)];
  }

  T query(int node, int nl, int nr, int ql, int qr) {
    propagation(node, nl, nr);

    if (ql > nr or qr < nl) return nq;

    if (ql <= nl and nr <= qr) return st[node];
  }
}

```



```

T x = query(left(node), nl, mid(nl, nr), ql, qr);
T y = query(right(node), mid(nl, nr) + 1, nr, ql, qr);

```

```

return x + y;
}

```

```

void propagation(int node, int nl, int nr) {
    if (lazy[node]) {
        st[node] += (nr - nl + 1) * lazy[node];

        if (nl < nr) {
            lazy[left(node)] += lazy[node];
            lazy[right(node)] += lazy[node];
        }

        lazy[node] = nu;
    }
}

int left(int p) { return p << 1; }
int right(int p) { return (p << 1) + 1; }
int mid(int l, int r) { return (r - l) / 2 + 1; }
};

```

## 1.15 Sparse Table

Answer the range query defined at the function `op`.

Build:  $O(N \log N)$ , Query:  $O(1)$

```

template <typename T>
struct SparseTable {
    vector<T> v;
    int n;
    static const int b = 30;
    vi mask, t;

    int op(int x, int y) { return v[x] < v[y] ? x : y; }
    int msb(int x) { return __builtin_clz(1) - __builtin_clz(x); }
    SparseTable() {}
    SparseTable(const vector<T>& v_) : v(v_), n(v.size()), mask(n), t(n) {
        for (int i = 0, at = 0; i < n; mask[i++] = at | = 1) {
            at = (at << 1) & ((1 << b) - 1);
            while (at and op(i, i - msb(at & -at)) == i) at ^= at & -at;
        }
        for (int i = 0; i < n / b; i++)
            t[i] = b * i + b - 1 - msb(mask[b * i + b - 1]);
        for (int j = 1; (1 << j) <= n / b; j++)
            for (int i = 0; i + (1 << j) <= n / b; i++)
                t[n / b * j + i] =
                    op(t[n / b * (j - 1) + i], t[n / b * (j - 1) + i + (1 << (j - 1))]);
    }
    int small(int r, int sz = b) { return r - msb(mask[r] & ((1 << sz) - 1)); }
    T query(int l, int r) {
        if (r - l + 1 <= b) return small(r, r - l + 1);
        int ans = op(small(l + b - 1), small(r));
        int x = l / b + 1, y = r / b - 1;
        if (x <= y) {
            int j = msb(y - x + 1);
            ans = op(ans, op(t[n / b * j + x], t[n / b * j + y - (1 << j) + 1]));

```

```

    }
    return ans;
}
};

```

## 2 Dynamic programming

### 2.1 Binary Knapsack (bottom up)

Given  $N$  items, each with its own value  $V_i$  and weight  $W_i$  and a maximum knapsack weight  $W$ , compute the maximum value of the items that we can carry, if we can either ignore or take a particular item.

Assume that  $1 \leq n \leq 1000$ ,  $1 \leq S \leq 10000$ .

Time and space:  $O(N * W)$

the vectors `VS` and `WS` starts at one, so it need an empty value at index 0.

```

const int MAXN{2010}, MAXM{2010};
ll st[MAXN][MAXM];
char ps[MAXN][MAXM];
pair<ll, vi> knapsack(int M, const vll &VS, const vi &WS) {
    memset(st, 0, sizeof(st));
    memset(ps, 0, sizeof(st));
    int N = len(VS) - 1; // ELEMENTS START AT INDEX 1 !

    for (int i = 0; i <= N; ++i) st[i][0] = 0;

    for (int m = 0; m <= M; ++m) st[0][m] = 0;

    for (int i = 1; i <= N; ++i) {
        for (int m = 1; m <= M; ++m) {
            st[i][m] = st[i - 1][m];
            ps[i][m] = 0;
            int w = WS[i];
            ll v = VS[i];

            if (w <= m and st[i - 1][m - w] + v > st[i][m]) {
                st[i][m] = st[i - 1][m - w] + v;
                ps[i][m] = 1;
            }
        }
    }

    int m = M;
    vi is;
    for (int i = N; i >= 1; --i) {
        if (ps[i][m]) {
            is.emplace_back(i - 1);
            m -= WS[i];
        }
    }

    return {st[N][M], is};
}

```

### 2.2 Binary Knapsack (top down)

Given  $N$  items, each with its own value  $V_i$  and weight  $W_i$  and a maximum knapsack weight  $W$ , compute the maximum value of the items that we can carry, if we can either ignore or take a particular item.

Assume that  $1 \leq n \leq 1000, 1 \leq S \leq 10000$ .

Time and space:  $O(N * W)$

the bottom up version is 5 times faster !

```
const int MAXN(2000), MAXM(2000);
ll memo[MAXN][MAXM + 1];
char choosen[MAXN][MAXM + 1];
ll knapSack(int u, int w, vll &VS, vi &WS) {
    if (u < 0) return 0;
    if (memo[u][w] != -1) return memo[u][w];

    ll a = 0, b = 0;
    a = knapSack(u - 1, w, VS, WS);
    if (WS[u] <= w) b = knapSack(u - 1, w - WS[u], VS, WS) + VS[u];
    if (b > a) {
        choosen[u][w] = true;
    }
    return memo[u][w] = max(a, b);
}

pair<ll, vi> knapSack(int W, vll &VS, vi &WS) {
    memset(memo, -1, sizeof(memo));
    memset(choosen, 0, sizeof(choosen));
    int n = len(VS);
    ll v = knapSack(n - 1, W, VS, WS);
    ll cw = W;
    vi choosed;
    for (int i = n - 1; i >= 0; i--) {
        if (choosen[i][cw]) {
            cw -= WS[i];
            choosed.emplace_back(i);
        }
    }
    return {v, choosed};
}
```

## 2.3 Edit Distance

$O(N * M)$

```
int edit_distance(const string &a, const string &b) {
    int n = a.size();
    int m = b.size();
    vector<vi> dp(n + 1, vi(m + 1, 0));

    int ADD = 1, DEL = 1, CHG = 1;
    for (int i = 0; i <= n; ++i) {
        dp[i][0] = i * DEL;
    }
    for (int i = 1; i <= m; ++i) {
        dp[0][i] = ADD * i;
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            int add = dp[i][j - 1] + ADD;
            int del = dp[i - 1][j] + DEL;
            int chg = dp[i - 1][j - 1] + (a[i - 1] == b[j - 1] ? 0 : 1) * CHG;
            dp[i][j] = min({add, del, chg});
        }
    }
}
```

```
    }
}

return dp[n][m];
}
```

## 2.4 Kadane

Find the maximum subarray sum in a given array.

```
int kadane(const vi &as) {
    vi s(len(as));
    s[0] = as[0];

    for (int i = 1; i < len(as); ++i) s[i] = max(as[i], s[i - 1] + as[i]);

    return *max_element(all(s));
}
```

## 2.5 Longest Increasing Subsequence (LIS)

Finds the length of the longest subsequence in

$O(n \log n)$

```
.

int LIS(const vi& as) {
    const ll oo = 1e18;
    int n = len(as);
    vll lis(n + 1, oo);
    lis[0] = -oo;

    auto ans = 0;

    for (int i = 0; i < n; ++i) {
        auto it = lower_bound(all(lis), as[i]);
        auto pos = (int)(it - lis.begin());

        ans = max(ans, pos);
        lis[pos] = as[i];
    }

    return ans;
}
```

## 2.6 Money Sum (Bottom Up)

Find every possible sum using the given values only once.

```
set<int> money_sum(const vi &xs) {
    using vc = vector<char>;
    using vvc = vector<vc>;
    int _m = accumulate(all(xs), 0);
    int _n = xs.size();
    vvc _dp(_n + 1, vc(_m + 1, 0));
    set<int> _ans;
    _dp[0][xs[0]] = 1;
    for (int i = 1; i < _n; ++i) {
        for (int j = 0; j <= _m; ++j) {
```

```

    if (j == 0 or _dp[i - 1][j]) {
        _dp[i][j + xs[i]] = 1;
        _dp[i][j] = 1;
    }
}
}

for (int i = 0; i < _n; ++i)
    for (int j = 0; j <= _m; ++j)
        if (_dp[i][j]) _ans.insert(j);
return _ans;
}

```

## 2.7 Travelling Salesman Problem

```

using vi = vector<int>;
vector<vi> dist;
vector<vi> memo;
/* 0 ( N^2 * 2^N ) */
int tsp(int i, int mask, int N) {
    if (mask == (1 << N) - 1) return dist[i][0];
    if (memo[i][mask] != -1) return memo[i][mask];
    int ans = INT_MAX << 1;
    for (int j = 0; j < N; ++j) {
        if (mask & (1 << j)) continue;
        auto t = tsp(j, mask | (1 << j), N) + dist[i][j];
        ans = min(ans, t);
    }
    return memo[i][mask] = ans;
}

```

## 3 Geometry

### 3.1 Convex Hull

Given a set of points find the smallest convex polygon that contains all the given points.

Time:  $O(N \log N)$

By default it removes the collinear points, set the boolean to true if you don't want that

```

struct pt {
    double x, y;
    int id;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

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

```

```

void convex_hull(vector<pt>& pts, bool include_collinear = false) {
    pt p0 = *min_element(all(pts), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(all(pts), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x - a.x) * (p0.x - a.x) + (p0.y - a.y) * (p0.y - a.y) <
                (p0.x - b.x) * (p0.x - b.x) + (p0.y - b.y) * (p0.y - b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = len(pts) - 1;
        while (i >= 0 && collinear(p0, pts[i], pts.back())) i--;
        reverse(pts.begin() + i + 1, pts.end());
    }

    vector<pt> st;
    for (int i = 0; i < len(pts); i++) {
        while (st.size() > 1 &&
            !cw(st[len(st) - 2], st.back(), pts[i], include_collinear))
            st.pop_back();
        st.push_back(pts[i]);
    }

    pts = st;
}

```

### 3.2 Determinant

```

#include "Point.cpp"

template <typename T>
T D(const Point<T> &P, const Point<T> &Q, const Point<T> &R) {
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
        (R.x * Q.y + R.y * P.x + Q.x * P.y);
}

```

### 3.3 Equals

```

template <typename T>
bool equals(T a, T b) {
    const double EPS{1e-9};
    if (is_floating_point<T>::value)
        return fabs(a - b) < EPS;
    else
        return a == b;
}

```

### 3.4 Line

```

#include <bits/stdc++.h>

#include "point-struct-and-utils.cpp"
using namespace std;

```

```

struct line {
    ld a, b, c;
};

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(const point &p1, const point &p2, line &l) {
    if (fabs(p1.x - p2.x) < EPS)
        // vertical line
        l = {1.0, 0.0, -p1.x};
    // default values
    else
        l = {-(ld)(p1.y - p2.y) / (p1.x - p2.x), 1.0, -(ld)(l.a * p1.x) - p1.y};
}

```

## 3.5 Point Struct And Utils (2d)

```

#include <bits/stdc++.h>
using namespace std;
using ld = long double;

struct point {
    ld x, y;
    int id;
    point(ld x = 0.0, ld y = 0.0, int id = -1) : x(x), y(y), id(id) {}

    point& operator+=(const point& t) {
        x += t.x;
        y += t.y;
        return *this;
    }

    point& operator-=(const point& t) {
        x -= t.x;
        y -= t.y;
        return *this;
    }

    point& operator*=(ld t) {
        x *= t;
        y *= t;
        return *this;
    }

    point& operator/=(ld t) {
        x /= t;
        y /= t;
        return *this;
    }

    point operator+(const point& t) const { return point(*this) += t; }
    point operator-(const point& t) const { return point(*this) -= t; }
    point operator*(ld t) const { return point(*this) *= t; }
    point operator/(ld t) const { return point(*this) /= t; }
};

ld dot(point& a, point& b) { return a.x * b.x + a.y * b.y; }

ld norm(point& a) { return dot(a, a); }

ld abs(point a) { return sqrt(norm(a)); }

ld proj(point a, point b) { return dot(a, b) / abs(b); }

```

```
ld angle(point a, point b) { return acos(dot(a, b) / abs(a) / abs(b)); }
```

```
ld cross(point a, point b) { return a.x * b.y - a.y * b.x; }
```

## 3.6 Segment

```

#include "Line.cpp"
#include "Point.cpp"
#include "equals.cpp"

```

```

template <typename T>
struct segment {
    Point<T> A, B;

    bool contains(const Point<T> &P) const;

    Point<T> closest(const Point<T> &p) const;
};

template <typename T>
bool segment<T>::contains(const Point<T> &P) const {
    // verifica se P áest contido na reta
    double dAB = Point<T>::dist(A, B), dAP = Point<T>::dist(A, P),
        dPB = Point<T>::dist(P, B);

    return equals(dAP + dPB, dAB);
}

template <typename T>
Point<T> segment<T>::closest(const Point<T> &P) const {
    Line<T> R(A, B);
    auto Q = R.closest(P);

    if (this->contains(Q)) return Q;

    auto distA = Point<T>::dist(P, A);
    auto distB = Point<T>::dist(P, B);

    if (distA <= distB)
        return A;
    else
        return B;
}

```

## 4 Graphs

### 4.1 2 SAT

```

struct SAT2 {
    ll n;
    vll2d adj, adj_t;
    vc used;
    vll order, comp;
    vc assignment;
    bool solvable;
    SAT2(ll _n)

```

```

: n(2 * _n),
  adj(n),
  adj_t(n),
  used(n),
  order(n),
  comp(n, -1),
  assignment(n / 2) {}
void dfs1(int v) {
  used[v] = true;
  for (int u : adj[v]) {
    if (!used[u]) dfs1(u);
  }
  order.push_back(v);
}

void dfs2(int v, int c1) {
  comp[v] = c1;
  for (int u : adj_t[v]) {
    if (comp[u] == -1) dfs2(u, c1);
  }
}

bool solve_2SAT() {
  // find and label each SCC
  for (int i = 0; i < n; ++i) {
    if (!used[i]) dfs1(i);
  }
  reverse(all(order));
  ll j = 0;
  for (auto &v : order) {
    if (comp[v] == -1) dfs2(v, j++);
  }

  assignment.assign(n / 2, false);
  for (int i = 0; i < n; i += 2) {
    // x and !x belong to the same SCC
    if (comp[i] == comp[i + 1]) {
      solvable = false;
      return false;
    }

    assignment[i / 2] = comp[i] > comp[i + 1];
  }
  solvable = true;
  return true;
}

void add_disjunction(int a, bool na, int b, bool nb) {
  a = (2 * a) ^ na;
  b = (2 * b) ^ nb;
  int neg_a = a ^ 1;
  int neg_b = b ^ 1;
  adj[neg_a].push_back(b);
  adj[neg_b].push_back(a);
  adj_t[b].push_back(neg_a);
  adj_t[a].push_back(neg_b);
}
};

```

## 4.2 Cycle Distances

Given a vertex  $s$  finds the longest cycle that end's in  $s$ , note that the vector **dist** will contain the distance that each vertex  $u$  needs to reach  $s$ .

Time:  $O(N)$

```

using adj = vector<vector<pair<int, ll>>>>;
ll cycleDistances(int u, int n, int s, vc &vis, adj &g, vll &dist) {
  vis[u] = 1;

  for (auto [v, d] : g[u]) {
    if (v == s) {
      dist[u] = max(dist[u], d);
      continue;
    }

    if (vis[v] == 1) {
      continue;
    }

    if (vis[v] == 2) {
      dist[u] = max(dist[u], dist[v] + d);
    } else {
      ll d2 = cycleDistances(v, n, s, vis, g, dist);
      if (d2 != -oo) {
        dist[u] = max(dist[u], d2 + d);
      }
    }
  }
  vis[u] = 2;
  return dist[u];
}

```

## 4.3 SCC (struct)

Able to find the component of each node and the total of SCC in  $O(V * E)$  and build the SCC graph ( $O(V * E)$ ).

```

struct SCC {
  ll N;
  int totsc;
  vll2d adj, tadj;
  vll todo, comps, comp;
  vector<set<ll>> sccadj;
  vchar vis;
  SCC(ll _N)
    : N(_N), totsc(0), adj(_N), tadj(_N), comp(_N, -1), sccadj(_N), vis(_N)
  {}

  void add_edge(ll x, ll y) { adj[x].eb(y), tadj[y].eb(x); }

  void dfs(ll x) {
    vis[x] = 1;
    for (auto &y : adj[x])
      if (!vis[y]) dfs(y);
    todo.pb(x);
  }

  void dfs2(ll x, ll v) {
    comp[x] = v;
  }
}

```

```

    for (auto &y : tadj[x])
        if (comp[y] == -1) dfs2(y, v);
}

void gen() {
    for (ll i = 0; i < N; ++i)
        if (!vis[i]) dfs(i);
    reverse(all(todo));
    for (auto &x : todo)
        if (comp[x] == -1) {
            dfs2(x, x);
            comps.pb(x);
            totsc++++;
        }
}

void genSCCGraph() {
    for (ll i = 0; i < N; ++i) {
        for (auto &j : adj[i]) {
            if (comp[i] != comp[j]) {
                sccadj[comp[i]].insert(comp[j]);
            }
        }
    }
}

};

```

## 4.4 Bellman-Ford (find negative cycle)

Given a directed graph find a negative cycle by running  $n$  iterations, and if the last one produces a relaxation than there is a cycle.

Time:  $O(V \cdot E)$

```

const ll oo = 2500 * 1e9;

using graph = vector<vector<pair<int, ll>>>;
vi negative_cycle(graph &g, int n) {
    vll d(n, oo);
    vi p(n, -1);
    int x = -1;
    d[0] = 0;
    for (int i = 0; i < n; i++) {
        x = -1;
        for (int u = 0; u < n; u++) {
            for (auto &[v, l] : g[u]) {
                if (d[u] + l < d[v]) {
                    d[v] = d[u] + l;
                    p[v] = u;
                    x = v;
                }
            }
        }
    }

    if (x == -1)
        return {};
    else {
        for (int i = 0; i < n; i++) x = p[x];
        vi cycle;
    }
}

```

```

    for (int v = x;; v = p[v]) {
        cycle.pb(v);
        if (v == x and len(cycle) > 1) break;
    }
    reverse(all(cycle));
    return cycle;
}
}

```

## 4.5 Bellman Ford

Find shortest path from a single source to all other nodes. Can detect negative cycles.

Time:  $O(V * E)$

```

bool bellman_ford(const vector<vector<pair<int, ll>>> &g, int s,
                  vector<ll> &dist) {
    int n = (int)g.size();
    dist.assign(n, LLONG_MAX);

    vector<int> count(n);
    vector<char> in_queue(n);
    queue<int> q;

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

    while (not q.empty()) {
        int cur = q.front();
        q.pop();
        in_queue[cur] = false;

        for (auto [to, w] : g[cur]) {
            if (dist[cur] + w < dist[to]) {
                dist[to] = dist[cur] + w;
                if (not in_queue[to]) {
                    q.push(to);
                    in_queue[to] = true;
                    count[to]++;
                    if (count[to] > n) return false;
                }
            }
        }
    }

    return true;
}

```

## 4.6 Binary Lifting

$far[h][i]$  = the node that is  $2^h$  distance from node  $i$

Build :  $O(N * \log N)$

sometimes is useful invert the order of loops

```

const int maxlog = 20;
int far[maxlog + 1][n + 1];
int n;

```

```

for (int h = 1; h <= maxlog; h++) {
    for (int i = 1; i <= n; i++) {
        far[h][i] = far[h - 1][far[h - 1][i]];
    }
}

```

## 4.7 Check Bipartite

$O(V)$

```

bool checkBipartite(const ll n, const vector<vll> &adj) {
    ll s = 0;
    queue<ll> q;
    q.push(s);
    vll color(n, INF);
    color[s] = 0;
    bool isBipartite = true;
    while (!q.empty() && isBipartite) {
        ll u = q.front();
        q.pop();
        for (auto &v : adj[u]) {
            if (color[v] == INF) {
                color[v] = 1 - color[u];
                q.push(v);
            } else if (color[v] == color[u]) {
                return false;
            }
        }
    }
    return true;
}

```

## 4.8 Dijkstra (k Shortest Paths)

```

const ll oo = 1e9 * 1e5 + 1;
using adj = vector<vector<pll>>;
vector<priority_queue<ll>> dijkstra(const vector<vector<pll>> &g, int n, int s
,
                                int k) {
    priority_queue<pll, vector<pll>, greater<pll>> pq;

    vector<priority_queue<ll>> dist(n);
    dist[0].emplace(0);
    pq.emplace(0, s);
    while (!pq.empty()) {
        auto [d1, v] = pq.top();
        pq.pop();

        if (not dist[v].empty() and dist[v].top() < d1) continue;

        for (auto [d2, u] : g[v]) {
            if (len(dist[u]) < k) {
                pq.emplace(d2 + d1, u);
                dist[u].emplace(d2 + d1);
            } else {
                if (dist[u].top() > d1 + d2) {

```

```

                    dist[u].pop();
                    dist[u].emplace(d1 + d2);
                    pq.emplace(d2 + d1, u);
                }
            }
        }
    }
    return dist;
}

```

## 4.9 Dijkstra (restore Path)

```

pair<vll, vi> dijkstra(const vector<vector<pll>> &g, int n, int s) {
    priority_queue<pll, vector<pll>, greater<pll>> pq;
    vll dist(n, oo);
    vi p(n, -1);
    pq.emplace(0, s);
    dist[s] = 0;
    while (!pq.empty()) {
        auto [d1, v] = pq.top();
        pq.pop();
        if (dist[v] < d1) continue;

        for (auto [d2, u] : g[v]) {
            if (dist[u] > d1 + d2) {
                dist[u] = d1 + d2;
                p[u] = v;
                pq.emplace(dist[u], u);
            }
        }
    }
    return {dist, p};
}

```

## 4.10 Dijkstra

Finds the minimum distance from  $s$  to every other node in

$$O(E * \log E)$$

time.

```

vll dijkstra(const vector<vector<pll>> &g, int n, int s) {
    priority_queue<pll, vector<pll>, greater<pll>> pq;
    vll dist(n + 1, oo);
    pq.emplace(0, s);
    dist[s] = 0;
    while (!pq.empty()) {
        auto [d1, v] = pq.top();
        pq.pop();
        if (dist[v] < d1) continue;

        for (auto [d2, u] : g[v]) {
            if (dist[u] > d1 + d2) {
                dist[u] = d1 + d2;
                pq.emplace(dist[u], u);
            }
        }
    }
}

```

```

    return dist;
}

```

## 4.11 Disjoint Edges Path (Maxflow)

Given a directed graph find's every path with distinct edges that starts at  $s$  and ends at  $t$

When building the graph, if there is an edge  $(u, v)$  is necessary to also add the transposed edge  $(v, u)$  but only need to add the capacity  $c(u, v)$ , and mark  $isedge(u, v)$  as true.

Time :  $O(E \cdot V^2)$

```

11 bfs(int s, int t, vi2d &g, vll2d &capacity, vi &parent) {
    fill(all(parent), -1);
    parent[s] = -2;
    queue<pair<ll, int>> q;
    q.push({0, s});

    while (!q.empty()) {
        auto [flow, cur] = q.front();
        q.pop();

        for (auto next : g[cur]) {
            if (parent[next] == -1 and capacity[cur][next]) {
                parent[next] = cur;
                ll new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({new_flow, next});
            }
        }
    }

    return 0;
}

11 maxflow(int s, int t, int n, vi2d &g, vll2d &capacity) {
    ll flow = 0;
    vi parent(n);
    ll new_flow;

    while ((new_flow = bfs(s, t, g, capacity, parent))) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}

void dfs(int u, int t, vi2d &g, vc2d &vis, vc2d &isedge, vll2d &capacity,
        vi &route, vi2d &routes) {
    route.pb(u);
    if (u == t) {
        routes.emplace_back(route);
        route.pop_back();
    }
}

```

```

    return;
}

for (auto &v : g[u]) {
    if (capacity[u][v] == 0 and isedge[u][v] and not vis[u][v]) {
        vis[u][v] = true;
        dfs(v, t, g, vis, isedge, capacity, route, routes);
        route.pop_back();
        return;
    }
}
}

vi2d disjoint_paths(vi2d &g, vll2d &capacity, vc2d &isedge, int s, int t,
                    int n) {
    ll mf = maxflow(s, t, n, g, capacity);
    vi2d routes;
    vi route;
    vc2d vis(n + 1, vc(n + 1));
    for (int i = 0; i < (int)mf; i++)
        dfs(s, t, g, vis, isedge, capacity, route, routes);
    return routes;
}

```

## 4.12 Euler Path (directed)

Given a **directed** graph finds a path that visits every edge exactly once.

Time:  $O(E)$

```

vector<int> euler_cycle(vector<vector<int>> &g, int u) {
    vector<int> res;

    stack<int> st;
    st.push(u);
    while (!st.empty()) {
        auto cur = st.top();
        if (g[cur].empty()) {
            res.push_back(cur);
            st.pop();
        } else {
            auto next = g[cur].back();
            st.push(next);

            g[cur].pop_back();
        }
    }

    for (auto &x : g)
        if (!x.empty()) return {};

    return res;
}

vector<int> euler_path(vector<vector<int>> &g, int first) {
    {
        int n = (int)g.size();
        vector<int> in(n), out(n);
        for (int i = 0; i < n; i++)
            for (auto x : g[i]) in[x]++, out[i]++;
    }
}

```



```

int a = 0, b = 0, c = 0;
for (int i = 0; i < n; i++)
    if (in[i] == out[i])
        c++;
    else if (in[i] - out[i] == 1)
        b++;
    else if (in[i] - out[i] == -1)
        a++;

if (c != n - 2 or a != 1 or b != 1) return {};
}

auto res = euler_cycle(g, first);
if (res.empty()) return res;

reverse(all(res));
return res;
}

```

### 4.13 Euler Path (undirected)

Given a **undirected** graph finds a path that visits every edge exactly once.  
Time:  $O(E)$

```

vector<int> euler_cycle(vector<vector<int>> &g, int u) {
    vector<int> res;
    multiset<pair<int, int>> vis;

    stack<int> st;
    st.push(u);
    while (!st.empty()) {
        auto cur = st.top();

        while (!g[cur].empty()) {
            auto it = vis.find(make_pair(cur, g[cur].back()));
            if (it == vis.end()) break;
            g[cur].pop_back();
            vis.erase(it);
        }

        if (g[cur].empty()) {
            res.push_back(cur);
            st.pop();
        } else {
            auto next = g[cur].back();
            st.push(next);

            vis.emplace(next, cur);
            g[cur].pop_back();
        }
    }

    for (auto &x : g)
        if (!x.empty()) return {};

    return res;
}

```

```

}

vector<int> euler_path(vector<vector<int>> &g, int first) {
    int n = (int)g.size();
    int v1 = -1, v2 = -1;
    {
        bool bad = false;
        for (int i = 0; i < n; i++)
            if (g[i].size() & 1) {
                if (v1 == -1)
                    v1 = i;
                else if (v2 == -1)
                    v2 = i;
                else
                    bad = true;
            }

        if (bad or (v1 != -1 and v2 == -1)) return {};
    }

    if (v2 != -1) {
        // insert cycle
        g[v1].push_back(v2);
        g[v2].push_back(v1);
    }

    auto res = euler_cycle(g, first);
    if (res.empty()) return res;

    if (v1 != -1) {
        for (int i = 0; i + 1 < (int)res.size(); i++) {
            if ((res[i] == v1 and res[i + 1] == v2) ||
                (res[i] == v2 and res[i + 1] == v1)) {
                vector<int> res2;
                for (int j = i + 1; j < (int)res.size(); j++) res2.push_back(res[j]);
                for (int j = 1; j <= i; j++) res2.push_back(res[j]);
                res = res2;
                break;
            }
        }
    }

    reverse(all(res));
    return res;
}

```

### 4.14 Find Centroid

Given a tree (don't forget to make it 'undirected'), find it's centroids.  
Time:  $O(V)$

```

void dfs(int u, int p, int n, vi2d &g, vi &sz, vi &centroid) {
    sz[u] = 1;

    bool iscentroid = true;
    for (auto v : g[u])
        if (v != p) {
            dfs(v, u, n, g, sz, centroid);

```

```

        if (sz[v] > n / 2) iscentroid = false;
        sz[u] += sz[v];
    }

    if (n - sz[u] > n / 2) iscentroid = false;
    if (iscentroid) centroid.eb(u);
}

vi getCentroid(vi2d &g, int n) {
    vi centroid;
    vi sz(n);
    dfs(0, -1, n, g, sz, centroid);
    return centroid;
}

```

## 4.15 Floyd Warshall

Simply finds the minimal distance for each node to every other node.  $O(V^3)$

```

vector<vll> floyd_warshall(const vector<vll> &adj, ll n) {
    auto dist = adj;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            for (int k = 0; k < n; ++k) {
                dist[j][k] = min(dist[j][k], dist[j][i] + dist[i][k]);
            }
        }
    }
    return dist;
}

```

## 4.16 Graph Cycle (directed)

Given a directed graph finds a cycle (or not).

Time:  $O(E)$

```

bool dfs(int v, vi2d &adj, vc &visited, vi &parent, vc &color, int &
    cycle_start,
    int &cycle_end) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
            if (dfs(u, adj, visited, parent, color, cycle_start, cycle_end))
                return true;
        } else if (color[u] == 1) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
    }
    color[v] = 2;
    return false;
}

vi find_cycle(vi2d &g, int n) {
    vc visited(n);

```

```

vi parent(n);
vc color(n);
int cycle_start, cycle_end;
color.assign(n, 0);
parent.assign(n, -1);
cycle_start = -1;

for (int v = 0; v < n; v++) {
    if (color[v] == 0 &&
        dfs(v, g, visited, parent, color, cycle_start, cycle_end))
        break;
}

if (cycle_start == -1) {
    return {};
} else {
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v]) cycle.push_back(v);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());
    return cycle;
}
}

```

## 4.17 Graph Cycle (undirected)

Detects if a graph contains a cycle. If path parameter is not null, it will contain the cycle if one exists.

Time:  $O(V + E)$

```

bool has_cycle(const vector<vector<int>> &g, int s, vector<char> &vis,
    vector<char> &in_path, vector<int> *path = nullptr) {
    vis[s] = in_path[s] = 1;
    if (path != nullptr) path->push_back(s);
    for (auto x : g[s]) {
        if (!vis[x] && has_cycle(g, x, vis, in_path, path))
            return true;
        else if (in_path[x]) {
            if (path != nullptr) path->push_back(x);
            return true;
        }
    }
    in_path[s] = 0;
    if (path != nullptr) path->pop_back();
    return false;
}

```

## 4.18 Kruskal

Find the minimum spanning tree of a graph.

Time:  $O(E \log E)$

can be used to find the maximum spanning tree by changing the comparison operator in the sort

```

struct UFDS {
    vector<int> ps, sz;
    int components;

```

```

UFDS(int n) : ps(n + 1), sz(n + 1, 1), components(n) { iota(all(ps), 0); }

int find_set(int x) { return (x == ps[x] ? x : (ps[x] = find_set(ps[x]))); }

bool same_set(int x, int y) { return find_set(x) == find_set(y); }

void union_set(int x, int y) {
    x = find_set(x);
    y = find_set(y);

    if (x == y) return;

    if (sz[x] < sz[y]) swap(x, y);

    ps[y] = x;
    sz[x] += sz[y];

    components--;
}

vector<tuple<ll, int, int>> kruskal(int n, vector<tuple<ll, int, int>> &edges)
{
    UFDS udfs(n);
    vector<tuple<ll, int, int>> ans;

    sort(all(edges));
    for (auto [a, b, c] : edges) {
        if (ufds.same_set(b, c)) continue;

        ans.emplace_back(a, b, c);
        udfs.union_set(b, c);
    }

    return ans;
}

```

## 4.19 Lowest Common Ancestor

Given two nodes of a tree find their lowest common ancestor, or their distance

Build :  $O(V)$ , Queries:  $O(1)$

**0 indexed !**

```

template <typename T>
struct SparseTable {
    vector<T> v;
    int n;
    static const int b = 30;
    vi mask, t;

    int op(int x, int y) { return v[x] < v[y] ? x : y; }
    int msb(int x) { return __builtin_clz(1) - __builtin_clz(x); }
    SparseTable() {}
    SparseTable(const vector<T>& v_) : v(v_), n(v.size()), mask(n), t(n) {
        for (int i = 0, at = 0; i < n; mask[i++] = at | = 1) {
            at = (at << 1) & ((1 << b) - 1);
            while (at and op(i, i - msb(at & -at)) == i) at ^= at & -at;
        }
    }
}

```

```

}
for (int i = 0; i < n / b; i++)
    t[i] = b * i + b - 1 - msb(mask[b * i + b - 1]);
for (int j = 1; (1 << j) <= n / b; j++)
    for (int i = 0; i + (1 << j) <= n / b; i++)
        t[n / b * j + i] =
            op(t[n / b * (j - 1) + i], t[n / b * (j - 1) + i + (1 << (j - 1))]);
}
int small(int r, int sz = b) { return r - msb(mask[r] & ((1 << sz) - 1)); }
T query(int l, int r) {
    if (r - l + 1 <= b) return small(r, r - l + 1);
    int ans = op(small(l + b - 1), small(r));
    int x = l / b + 1, y = r / b - 1;
    if (x <= y) {
        int j = msb(y - x + 1);
        ans = op(ans, op(t[n / b * j + x], t[n / b * j + y - (1 << j) + 1]));
    }
    return ans;
}
};

struct LCA {
    SparseTable<int> st;
    int n;
    vi v, pos, dep;

    LCA(const vi2d& g, int root) : n(len(g)), pos(n) {
        dfs(root, 0, -1, g);
        st = SparseTable<int>(vector<int>(all(dep)));
    }

    void dfs(int i, int d, int p, const vi2d& g) {
        v.eb(len(dep)) = i, pos[i] = len(dep), dep.eb(d);
        for (auto j : g[i])
            if (j != p) {
                dfs(j, d + 1, i, g);
                v.eb(len(dep)) = i, dep.eb(d);
            }
    }

    int lca(int a, int b) {
        int l = min(pos[a], pos[b]);
        int r = max(pos[a], pos[b]);
        return v[st.query(l, r)];
    }

    int dist(int a, int b) {
        return dep[pos[a]] + dep[pos[b]] - 2 * dep[pos[lca(a, b)]];
    }
};

```

## 4.20 Tree Maximum Distance

Returns the maximum distance from every node to any other node in the tree.  $O(6V) = O(V)$

```

pll mostDistantFrom(const vector<vll> &adj, ll n, ll root) {
    // O(V)
    // 0 indexed
    ll mostDistantNode = root;
}

```

```

    ll nodeDistance = 0;
    queue<pll> q;
    vector<char> vis(n);
    q.emplace(root, 0);
    vis[root] = true;
    while (!q.empty()) {
        auto [node, dist] = q.front();
        q.pop();
        if (dist > nodeDistance) {
            nodeDistance = dist;
            mostDistantNode = node;
        }
        for (auto u : adj[node]) {
            if (!vis[u]) {
                vis[u] = true;
                q.emplace(u, dist + 1);
            }
        }
    }
    return {mostDistantNode, nodeDistance};
}

ll twoNodesDist(const vector<vll> &adj, ll n, ll a, ll b) {
    queue<pll> q;
    vector<char> vis(n);
    q.emplace(a, 0);
    while (!q.empty()) {
        auto [node, dist] = q.front();
        q.pop();
        if (node == b) return dist;
        for (auto u : adj[node]) {
            if (!vis[u]) {
                vis[u] = true;
                q.emplace(u, dist + 1);
            }
        }
    }
    return -1;
}

tuple<ll, ll, ll> tree_diameter(const vector<vll> &adj, ll n) {
    // returns two points of the diameter and the diameter itself
    auto [node1, dist1] = mostDistantFrom(adj, n, 0); // O(V)
    auto [node2, dist2] = mostDistantFrom(adj, n, node1); // O(V)
    auto diameter = twoNodesDist(adj, n, node1, node2); // O(V)
    return make_tuple(node1, node2, diameter);
}

vll everyDistanceFromNode(const vector<vll> &adj, ll n, ll root) {
    // Single Source Shortest Path, from a given root
    queue<pair<ll, ll>> q;
    vll ans(n, -1);
    ans[root] = 0;
    q.emplace(root, 0);
    while (!q.empty()) {
        auto [u, d] = q.front();
        q.pop();

```

```

        for (auto w : adj[u]) {
            if (ans[w] != -1) continue;
            ans[w] = d + 1;
            q.emplace(w, d + 1);
        }
    }
    return ans;
}

vll maxDistances(const vector<vll> &adj, ll n) {
    auto [node1, node2, diameter] = tree_diameter(adj, n); // O(3V)
    auto distances1 = everyDistanceFromNode(adj, n, node1); // O(V)
    auto distances2 = everyDistanceFromNode(adj, n, node2); // O(V)
    vll ans(n);
    for (int i = 0; i < n; ++i)
        ans[i] = max(distances1[i], distances2[i]); // O(V)
    return ans;
}

```

## 4.21 Maximum Flow (Edmonds-Karp)

Finds the **maximum flow** in a graph network, given the **source**  $s$  and the **sink**  $t$ .

When building the graph, if there is an edge  $(u, v)$  is necessary to also add the transposed edge  $(v, u)$  but only need to add the capacity  $c(u, v)$ .

Time:  $O(V \cdot E^2)$

```

const ll oo = 1e17;

ll bfs(int s, int t, vi2d &g, vll2d &capacity, vi &parent) {
    fill(all(parent), -1);
    parent[s] = -2;
    queue<pair<ll, int>> q;
    q.push({oo, s});

    while (!q.empty()) {
        auto [flow, cur] = q.front();
        q.pop();

        for (auto next : g[cur]) {
            if (parent[next] == -1 and capacity[cur][next]) {
                parent[next] = cur;
                ll new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({new_flow, next});
            }
        }
    }

    return 0ll;
}

ll maxflow(int s, int t, int n, vi2d &g, vll2d &capacity) {
    ll flow = 0;
    vi parent(n);
    ll new_flow;

    while ((new_flow = bfs(s, t, g, capacity, parent))) {
        flow += new_flow;
    }
}

```

```

    int cur = t;
    while (cur != s) {
        int prev = parent[cur];
        capacity[prev][cur] -= new_flow;
        capacity[cur][prev] += new_flow;
        cur = prev;
    }
}

return flow;
}

```

## 4.22 Minimum Cut (unweighted)

Given the edges of a directed/undirected graph find the minum of edges that needs to be removed to make the sink  $t$  unreachable from the source  $s$ .

Time:  $O(V \cdot E^2)$

```

const ll oo = 1e17;

ll bfs(int s, int t, vi2d &g, vll2d &capacity, vi &parent) {
    fill(all(parent), -1);
    parent[s] = -2;
    queue<pair<ll, int>> q;
    q.push({oo, s});

    while (!q.empty()) {
        auto [flow, cur] = q.front();
        q.pop();

        for (auto next : g[cur]) {
            if (parent[next] == -1 and capacity[cur][next]) {
                parent[next] = cur;
                ll new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({new_flow, next});
            }
        }
    }

    return 0ll;
}

ll maxflow(int s, int t, int n, vi2d &g, vll2d &capacity) {
    ll flow = 0;
    vi parent(n);
    ll new_flow;

    while ((new_flow = bfs(s, t, g, capacity, parent))) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
}

```

```

    return flow;
}

void dfs(int u, vi2d &g, vll2d &capacity, vc &visited) {
    visited[u] = true;

    for (auto v : g[u]) {
        if (capacity[u][v] > 0 and not visited[v]) {
            dfs(v, g, capacity, visited);
        }
    }
}

vii mincut(vii &edges, int s, int t, int n, bool directed = false) {
    vll2d capacity(n, vll(n));
    vi2d g(n);
    for (auto &[u, v] : edges) {
        g[u].eb(v);
        capacity[u][v] += 1;
        if (not directed) {
            g[v].eb(u);
            capacity[v][u] += 1;
        }
    }

    maxflow(0, n - 1, n, g, capacity);
    vc vis(n);
    dfs(0, g, capacity, vis);

    vii removed;
    for (auto &[u, v] : edges) {
        if ((vis[u] and not vis[v]) or (vis[v] and not vis[u]))
            removed.emplace_back(u, v);
    }

    return removed;
}

```

## 4.23 Small to Large

Answer queries of the form "How many vertices in the subtree of vertex  $v$  have property  $P$ ?"

Build:  $O(N)$ , Query:  $O(N \log N)$

```

struct SmallToLarge {
    vector<vector<int>> tree, vis_chlds;
    vector<int> sizes, values, ans;
    set<int> cnt;

    SmallToLarge(vector<vector<int>> &&g, vector<int> &&v)
        : tree(g), vis_chlds(g.size()), sizes(g.size()), values(v), ans(g.size())
        {
            update_sizes(0);
        }

    inline void add_value(int u) { cnt.insert(values[u]); }

    inline void remove_value(int u) { cnt.erase(values[u]); }
}

```

```

inline void update_ans(int u) { ans[u] = (int)cnt.size(); }

void dfs(int u, int p = -1, bool keep = true) {
    int mx = -1;
    for (auto x : tree[u]) {
        if (x == p) continue;

        if (mx == -1 or sizes[mx] < sizes[x]) mx = x;

        for (auto x : tree[u]) {
            if (x != p and x != mx) dfs(x, u, false);
        }

        if (mx != -1) {
            dfs(mx, u, true);
            swap(vis_chlds[u], vis_chlds[mx]);
        }

        vis_chlds[u].push_back(u);
        add_value(u);

        for (auto x : tree[u]) {
            if (x != p and x != mx) {
                for (auto y : vis_chlds[x]) {
                    add_value(y);
                    vis_chlds[u].push_back(y);
                }
            }
        }

        update_ans(u);

        if (!keep) {
            for (auto x : vis_chlds[u]) remove_value(x);
        }
    }
}

void update_sizes(int u, int p = -1) {
    sizes[u] = 1;
    for (auto x : tree[u]) {
        if (x != p) {
            update_sizes(x, u);
            sizes[u] += sizes[x];
        }
    }
}
};

```

## 4.24 Sum every node distance

Given a **tree**, for each node  $i$  find the sum of distance from  $i$  to every other node.

**don't forget to set the tree as undirected, that's needed to choose an arbitrary root**

Time:  $O(N)$

```

void getRoot(int u, int p, vi2d &g, vll &d, vll &cnt) {
    for (int i = 0; i < len(g[u]); i++) {

```

```

        int v = g[u][i];
        if (v == p) continue;
        getRoot(v, u, g, d, cnt);
        d[u] += d[v] + cnt[v];
        cnt[u] += cnt[v];
    }
}

void dfs(int u, int p, vi2d &g, vll &cnt, vll &ansd, int n) {
    for (int i = 0; i < len(g[u]); i++) {
        int v = g[u][i];
        if (v == p) continue;

        ansd[v] = ansd[u] - cnt[v] + (n - cnt[v]);
        dfs(v, u, g, cnt, ansd, n);
    }
}

vll fromToAll(vi2d &g, int n) {
    vll d(n);
    vll cnt(n, 1);
    getRoot(0, -1, g, d, cnt);

    vll ansdist(n);
    ansdist[0] = d[0];

    dfs(0, -1, g, cnt, ansdist, n);
    return ansdist;
}

```

## 4.25 Topological Sorting

Assumes that :

- vertices index  $[0, n - 1]$
- is a DAG (else it returns an empty vector)

$O(V)$

```

enum class state { not_visited, processing, done };
bool dfs(const vector<vll> &adj, ll s, vector<state> &states, vll &order) {
    states[s] = state::processing;
    for (auto &v : adj[s]) {
        if (states[v] == state::not_visited) {
            if (not dfs(adj, v, states, order)) return false;
        } else if (states[v] == state::processing)
            return false;
    }
    states[s] = state::done;
    order.pb(s);
    return true;
}

vll topologicalSorting(const vector<vll> &adj) {
    ll n = len(adj);
    vll order;
    vector<state> states(n, state::not_visited);
    for (int i = 0; i < n; ++i) {
        if (states[i] == state::not_visited) {
            if (not dfs(adj, i, states, order)) return {};
        }
    }
}

```

```

}
reverse(all(order));
return order;
}

```

## 4.26 Tree Diameter

Finds the length of the diameter of the tree in  $O(V)$ , it's easy to recover the nodes at the point of the diameter .

```

pll mostDistantFrom(const vector<vll> &adj, ll n, ll root) {
    // 0 indexed
    ll mostDistantNode = root;
    ll nodeDistance = 0;
    queue<pll> q;
    vector<char> vis(n);
    q.emplace(root, 0);
    vis[root] = true;
    while (!q.empty()) {
        auto [node, dist] = q.front();
        q.pop();
        if (dist > nodeDistance) {
            nodeDistance = dist;
            mostDistantNode = node;
        }
        for (auto u : adj[node]) {
            if (!vis[u]) {
                vis[u] = true;
                q.emplace(u, dist + 1);
            }
        }
    }
    return {mostDistantNode, nodeDistance};
}

ll twoNodesDist(const vector<vll> &adj, ll n, ll a, ll b) {
    // 0 indexed
    queue<pll> q;
    vector<char> vis(n);
    q.emplace(a, 0);
    while (!q.empty()) {
        auto [node, dist] = q.front();
        q.pop();
        if (node == b) {
            return dist;
        }
        for (auto u : adj[node]) {
            if (!vis[u]) {
                vis[u] = true;
                q.emplace(u, dist + 1);
            }
        }
    }
    return -1;
}

ll tree_diameter(const vector<vll> &adj, ll n) {
    // 0 indexed !!!
    auto [node1, dist1] = mostDistantFrom(adj, n, 0); // 0(V)
    auto [node2, dist2] = mostDistantFrom(adj, n, node1); // 0(V)
}

```

```

auto diameter = twoNodesDist(adj, n, node1, node2); // 0(V)
return diameter;
}

```

## 5 Math

### 5.1 GCD (with factorization)

$O(\sqrt{n})$  due to factorization.

```

ll gcd_with_factorization(ll a, ll b) {
    map<ll, ll> fa = factorization(a);
    map<ll, ll> fb = factorization(b);
    ll ans = 1;
    for (auto fai : fa) {
        ll k = min(fai.second, fb[fai.first]);
        while (k--) ans *= fai.first;
    }
    return ans;
}

```

### 5.2 GCD

```

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

```

### 5.3 LCM (with factorization)

$O(\sqrt{n})$  due to factorization.

```

ll lcm_with_factorization(ll a, ll b) {
    map<ll, ll> fa = factorization(a);
    map<ll, ll> fb = factorization(b);
    ll ans = 1;
    for (auto fai : fa) {
        ll k = max(fai.second, fb[fai.first]);
        while (k--) ans *= fai.first;
    }
    return ans;
}

```

### 5.4 LCM

```

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

```

### 5.5 Arithmetic Progression Sum

- $s$  : first term
- $d$  : common difference
- $n$  : number of terms

```

ll arithmeticProgressionSum(ll s, ll d, ll n) {
    return (s + (s + d * (n - 1))) * n / 2ll;
}

```

## 5.6 Binomial MOD

Precompute every factorial until  $maxn$  ( $O(maxn)$ ) allowing to answer the  $\binom{n}{k}$  in  $O(\log mod)$  time, due to the fastpow. Note that it needs  $O(maxn)$  in memory.

```
const ll MOD = 1e9 + 7;
const ll maxn = 2 * 1e6;
vll fats(maxn + 1, -1);
void precompute() {
    fats[0] = 1;
    for (ll i = 1; i <= maxn; i++) {
        fats[i] = (fats[i - 1] * i) % MOD;
    }
}

ll fpow(ll a, ll n, ll mod = LLONG_MAX) {
    if (n == 0ll) return 1ll;
    if (n == 1ll) return a;
    ll x = fpow(a, n / 2ll, mod) % mod;
    return ((x * x) % mod * (n & 1ll ? a : 1ll)) % mod;
}

ll binommod(ll n, ll k) {
    ll upper = fats[n];
    ll lower = (fats[k] * fats[n - k]) % MOD;
    return (upper * fpow(lower, MOD - 2ll, MOD)) % MOD;
}
```

## 5.7 Binomial

$O(nm)$  time,  $O(m)$  space  
Equal to  $n$  choose  $k$

```
ll binom(ll n, ll k) {
    if (k > n) return 0;
    vll dp(k + 1, 0);
    dp[0] = 1;
    for (ll i = 1; i <= n; i++)
        for (ll j = k; j > 0; j--) dp[j] = dp[j] + dp[j - 1];
    return dp[k];
}
```

## 5.8 Euler phi $\varphi(n)$ (in range)

Computes the number of positive integers less than  $n$  that are coprimes with  $n$ , in the range  $[1, n]$ , in  $O(N \log N)$ .

```
const int MAX = 1e6;
vi range_phi(int n) {
    bitset<MAX> sieve;
    vi phi(n + 1);

    iota(phi.begin(), phi.end(), 0);
    sieve.set();

    for (int p = 2; p <= n; p += 2) phi[p] /= 2;
    for (int p = 3; p <= n; p += 2) {
        if (sieve[p]) {
```

```
            for (int j = p; j <= n; j += p) {
                sieve[j] = false;
                phi[j] /= p;
                phi[j] *= (p - 1);
            }
        }
    }

    return phi;
}
```

## 5.9 Euler phi $\varphi(n)$

Computes the number of positive integers less than  $n$  that are coprimes with  $n$ , in  $O(\sqrt{N})$ .

```
int phi(int n) {
    if (n == 1) return 1;

    auto fs = factorization(n); // a vector of pair or a map
    auto res = n;

    for (auto [p, k] : fs) {
        res /= p;
        res *= (p - 1);
    }

    return res;
}
```

## 5.10 Factorial Factorization

Computes the factorization of  $n!$  in  $\pi(N) * \log n$

```
// O(logN)
ll E(ll n, ll p) {
    ll k = 0, b = p;
    while (b <= n) {
        k += n / b;
        b *= p;
    }
    return k;
}

// O(pi(N)*logN)
map<ll, ll> factorial_factorization(ll n, const vll &primes) {
    map<ll, ll> fs;
    for (const auto &p : primes) {
        if (p > n) break;
        fs[p] = E(n, p);
    }
    return fs;
}
```



## 5.11 Factorial

```
const ll MAX = 18;
vll fv(MAX, -1);
ll factorial(ll n) {
    if (fv[n] != -1) return fv[n];
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

## 5.12 Factorization (Pollard Rho)

Factorizes a number into its prime factors in  $O(n^{\frac{1}{4}} * \log(n))$ .

```
ll mul(ll a, ll b, ll m) {
    ll ret = a * b - (ll)((ld)1 / m * a * b + 0.5) * m;
    return ret < 0 ? ret + m : ret;
}

ll pow(ll a, ll b, ll m) {
    ll ans = 1;
    for (; b > 0; b /= 2ll, a = mul(a, a, m)) {
        if (b % 2ll == 1) ans = mul(ans, a, m);
    }
    return ans;
}

bool prime(ll n) {
    if (n < 2) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0) return 0;

    ll r = __builtin_ctzll(n - 1), d = n >> r;
    for (int a : {2, 325, 9375, 28178, 450775, 9780504, 795265022}) {
        ll x = pow(a, d, n);
        if (x == 1 or x == n - 1 or a % n == 0) continue;

        for (int j = 0; j < r - 1; j++) {
            x = mul(x, x, n);
            if (x == n - 1) break;
        }
        if (x != n - 1) return 0;
    }
    return 1;
}

ll rho(ll n) {
    if (n == 1 or prime(n)) return n;
    auto f = [n](ll x) { return mul(x, x, n) + 1; };

    ll x = 0, y = 0, t = 30, prd = 2, x0 = 1, q;
    while (t % 40 != 0 or gcd(prd, n) == 1) {
        if (x == y) x = ++x0, y = f(x);
        q = mul(prd, abs(x - y), n);
        if (q != 0) prd = q;
        x = f(x), y = f(f(y)), t++;
    }
    return gcd(prd, n);
}
```

```
}

vll fact(ll n) {
    if (n == 1) return {};
    if (prime(n)) return {n};
    ll d = rho(n);
    vll l = fact(d), r = fact(n / d);
    l.insert(l.end(), r.begin(), r.end());
    return l;
}
```

## 5.13 Factorization

Computes the factorization of n in  $O(\sqrt{n})$ .

```
map<ll, ll> factorization(ll n) {
    map<ll, ll> ans;
    for (ll i = 2; i * i <= n; i++) {
        ll count = 0;
        for (; n % i == 0; count++, n /= i)
            ;
        if (count) ans[i] = count;
    }
    if (n > 1) ans[n]++;
    return ans;
}
```

## 5.14 Fast Fourier Transform

```
template <bool invert = false>
void fft(vector<complex<double>>& xs) {
    int N = (int)xs.size();

    if (N == 1) return;

    vector<complex<double>> es(N / 2), os(N / 2);

    for (int i = 0; i < N / 2; ++i) es[i] = xs[2 * i];

    for (int i = 0; i < N / 2; ++i) os[i] = xs[2 * i + 1];

    fft<invert>(es);
    fft<invert>(os);

    auto signal = (invert ? 1 : -1);
    auto theta = 2 * signal * acos(-1) / N;
    complex<double> S{1}, S1{cos(theta), sin(theta)};

    for (int i = 0; i < N / 2; ++i) {
        xs[i] = (es[i] + S * os[i]);
        xs[i] /= (invert ? 2 : 1);

        xs[i + N / 2] = (es[i] - S * os[i]);
        xs[i + N / 2] /= (invert ? 2 : 1);

        S *= S1;
    }
}
```

```

}
}

```

## 5.15 Fast pow

Computes  $a^n$  in  $O(\log N)$ .

```

ll fpow(ll a, int n, ll mod = LLONG_MAX) {
    if (n == 0) return 1;
    if (n == 1) return a;
    ll x = fpow(a, n / 2, mod) % mod;
    return ((x * x) % mod * (n & 1 ? a : 1)) % mod;
}

```

## 5.16 Gauss Elimination

```

template <size_t Dim>
struct GaussianElimination {
    vector<ll> basis;
    size_t size;

    GaussianElimination() : basis(Dim + 1), size(0) {}

    void insert(ll x) {
        for (ll i = Dim; i >= 0; i--) {
            if ((x & 1ll << i) == 0) continue;

            if (!basis[i]) {
                basis[i] = x;
                size++;
                break;
            }

            x ^= basis[i];
        }
    }

    void normalize() {
        for (ll i = Dim; i >= 0; i--)
            for (ll j = i - 1; j >= 0; j--)
                if (basis[i] & 1ll << j) basis[i] ^= basis[j];
    }

    bool check(ll x) {
        for (ll i = Dim; i >= 0; i--) {
            if ((x & 1ll << i) == 0) continue;

            if (!basis[i]) return false;

            x ^= basis[i];
        }

        return true;
    }

    auto operator[](ll k) { return at(k); }
}

```

```

ll at(ll k) {
    ll ans = 0;
    ll total = 1ll << size;
    for (ll i = Dim; ~i; i--) {
        if (!basis[i]) continue;

        ll mid = total >> 1ll;
        if ((mid < k and (ans & 1ll << i) == 0) ||
            (k <= mid and (ans & 1ll << i)))
            ans ^= basis[i];

        if (mid < k) k -= mid;

        total >>= 1ll;
    }
    return ans;
}

ll at_normalized(ll k) {
    ll ans = 0;
    k--;
    for (size_t i = 0; i <= Dim; i++) {
        if (!basis[i]) continue;
        if (k & 1) ans ^= basis[i];
        k >>= 1;
    }
    return ans;
}
};

```

## 5.17 Integer Mod

```

const ll INF = 1e18;
const ll mod = 998244353;
template <ll MOD = mod>
struct Modular {
    ll value;
    static const ll MOD_value = MOD;

    Modular(ll v = 0) {
        value = v % MOD;
        if (value < 0) value += MOD;
    }

    Modular(ll a, ll b) : value(0) {
        *this += a;
        *this /= b;
    }

    Modular& operator+=(Modular const& b) {
        value += b.value;
        if (value >= MOD) value -= MOD;
        return *this;
    }

    Modular& operator-=(Modular const& b) {
        value -= b.value;
        if (value < 0) value += MOD;
        return *this;
    }
}

```

```

Modular& operator==(Modular const& b) {
    value = (ll)value * b.value % MOD;
    return *this;
}

friend Modular mexp(Modular a, ll e) {
    Modular res = 1;
    while (e) {
        if (e & 1) res *= a;
        a *= a;
        e >>= 1;
    }
    return res;
}

friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }

Modular& operator/=(Modular const& b) { return *this *= inverse(b); }
friend Modular operator+(Modular a, Modular const b) { return a += b; }
Modular operator++(int) { return this->value = (this->value + 1) % MOD; }
Modular operator++() { return this->value = (this->value + 1) % MOD; }
friend Modular operator-(Modular a, Modular const b) { return a -= b; }
friend Modular operator-(Modular const a) { return 0 - a; }
Modular operator--(int) {
    return this->value = (this->value - 1 + MOD) % MOD;
}

Modular operator--() { return this->value = (this->value - 1 + MOD) % MOD; }
friend Modular operator*(Modular a, Modular const b) { return a *= b; }
friend Modular operator/(Modular a, Modular const b) { return a /= b; }
friend std::ostream& operator<<(std::ostream& os, Modular const& a) {
    return os << a.value;
}

friend bool operator==(Modular const& a, Modular const& b) {
    return a.value == b.value;
}

friend bool operator!=(Modular const& a, Modular const& b) {
    return a.value != b.value;
}
};

```

## 5.18 Is prime

$O(\sqrt{N})$

```

bool isprime(ll n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    for (ll i = 3; i * i < n; i += 2)
        if (n % i == 0) return false;
    return true;
}

```

## 5.19 Number of Divisors $\tau(n)$

Find the total of divisors of  $N$  in  $O(\sqrt{N})$

```

ll number_of_divisors(ll n) {

```

```

    ll res = 0;

    for (ll d = 1; d * d <= n; ++d) {
        if (n % d == 0) res += (d == n / d ? 1 : 2);
    }

    return res;
}

```

## 5.20 Power Sum

Calculates  $K^0 + K^1 + \dots + K^n$

```

ll powersum(ll n, ll k) { return (fastpow(n, k + 1) - 1) / (k - 1); }

```

## 5.21 Sieve list primes

List every prime until MAXN,  $O(N \log N)$  in time and  $O(MAXN)$  in memory.

```

const ll MAXN = 1e5;
vll list_primes(ll n) {
    vll ps;
    bitset<MAXN> sieve;
    sieve.set();
    sieve.reset(1);
    for (ll i = 2; i <= n; ++i) {
        if (sieve[i]) ps.push_back(i);
        for (ll j = i * 2; j <= n; j += i) {
            sieve.reset(j);
        }
    }
    return ps;
}

```

## 5.22 Sum of Divisors $\sigma(n)$

Computes the sum of each divisor of  $n$  in  $O(\sqrt{n})$ .

```

ll sum_of_divisors(long long n) {
    ll res = 0;

    for (ll d = 1; d * d <= n; ++d) {
        if (n % d == 0) {
            ll k = n / d;

            res += (d == k ? d : d + k);
        }
    }

    return res;
}

```

## 6 Problems

### 6.1 Hanoi Tower

Let  $T_n$  be the total of moves to solve a hanoi tower, we know that  $T_n \geq 2 \cdot T_{n-1} + 1$ , for  $n > 0$ , and  $T_0 = 0$ . By induction it's easy to see that  $T_n = 2^n - 1$ , for  $n > 0$ .

The following algorithm finds the necessary steps to solve the game for 3 stacks and  $n$  disks.

```
void move(int a, int b) { cout << a << ' ' << b << endl; }
void solve(int n, int s, int e) {
    if (n == 0) return;
    if (n == 1) {
        move(s, e);
        return;
    }
    solve(n - 1, s, 6 - s - e);
    move(s, e);
    solve(n - 1, 6 - s - e, e);
}
```

## 7 Searching

### 7.1 Meet in the middle

Answers the query how many subsets of the vector  $xs$  have sum equal  $x$ .

Time:  $O(N \cdot 2^{\frac{N}{2}})$

```
vll get_subset_sums(int l, int r, vll &a) {
    int len = r - l + 1;
    vll res;

    for (int i = 0; i < (1 << len); i++) {
        ll sum = 0;
        for (int j = 0; j < len; j++) {
            if (i & (1 << j)) {
                sum += a[l + j];
            }
        }
        res.push_back(sum);
    }
    return res;
};

ll count(vll &xs, ll x) {
    int n = len(xs);
    vll left = get_subset_sums(0, n / 2 - 1, xs);
    vll right = get_subset_sums(n / 2, n - 1, xs);
    sort(all(left));
    sort(all(right));
    ll ans = 0;
    for (ll i : left) {
        auto start_index =
            lower_bound(right.begin(), right.end(), x - i) - right.begin();
        auto end_index =
            upper_bound(right.begin(), right.end(), x - i) - right.begin();
        ans += end_index - start_index;
    }
}
```

```
return ans;
}
```

### 7.2 Ternary Search Recursive

```
const double eps = 1e-6;

// IT MUST BE AN UNIMODAL FUNCTION
double f(int x) { return x * x + 2 * x + 4; }

double ternary_search(double l, double r) {
    if (fabs(f(l) - f(r)) < eps) return f((l + (r - l) / 2.0));

    auto third = (r - l) / 3.0;
    auto m1 = l + third;
    auto m2 = r - third;

    // change the signal to find the maximum point.
    return m1 < m2 ? ternary_search(m1, r) : ternary_search(l, m2);
}
```

## 8 Strings

### 8.1 Count Distinct Anagrams

```
const ll MOD = 1e9 + 7;
const int maxn = 1e6;
vll fs(maxn + 1);
void precompute() {
    fs[0] = 1;
    for (ll i = 1; i <= maxn; i++) {
        fs[i] = (fs[i - 1] * i) % MOD;
    }
}

ll fpow(ll a, int n, ll mod = LLONG_MAX) {
    if (n == 0) return 1;
    if (n == 1) return a;
    ll x = fpow(a, n / 2, mod) % mod;
    return ((x * x) % mod * (n & 1 ? a : 1)) % mod;
}

ll distinctAnagrams(const string &s) {
    precompute();
    vi hist('z' - 'a' + 1, 0);
    for (auto &c : s) hist[c - 'a']++;
    ll ans = fs[len(s)];
    for (auto &q : hist) {
        ans = (ans * fpow(fs[q], MOD - 2, MOD)) % MOD;
    }
    return ans;
}
```

### 8.2 Double Hash Range Query

```
const ll MOD = 1000027957;
const int MOD2 = 1000015187;
```

```
struct Hash {
    const ll P = 31;
    int n;
    string s;
    vll h, h2, hi, hi2, p, p2;
    Hash() {}
    Hash(string _s) : s(_s), n(len(_s)), h(n), h2(n), hi(n), hi2(n), p(n), p2(n)
    {
        for (int i = 0; i < n; i++) p[i] = (i ? P * p[i - 1] : 1) % MOD;
        for (int i = 0; i < n; i++) p2[i] = (i ? P * p2[i - 1] : 1) % MOD2;
        for (int i = 0; i < n; i++) h[i] = (s[i] + (i ? h[i - 1] : 0) * P) % MOD;
        for (int i = 0; i < n; i++) h2[i] = (s[i] + (i ? h2[i - 1] : 0) * P) %
        MOD2;
        for (int i = n - 1; i >= 0; i--)
            hi[i] = (s[i] + (i + 1 < n ? hi[i + 1] : 0) * P) % MOD;
        for (int i = n - 1; i >= 0; i--)
            hi2[i] = (s[i] + (i + 1 < n ? hi2[i + 1] : 0) * P) % MOD2;
    }
    pii query(int l, int r) {
        ll hash = (h[r] - (l ? h[l - 1] * p[r - l + 1] % MOD : 0));
        ll hash2 = (h2[r] - (l ? h2[l - 1] * p2[r - l + 1] % MOD2 : 0));

        return {(hash < 0 ? hash + MOD : hash), (hash2 < 0 ? hash2 + MOD2 : hash2)};
    }
    pii query_inv(int l, int r) {
        ll hash = (hi[l] - (r + 1 < n ? hi[r + 1] * p[r - l + 1] % MOD : 0));
        ll hash2 = (hi2[l] - (r + 1 < n ? hi2[r + 1] * p2[r - l + 1] % MOD2 : 0));
        return {(hash < 0 ? hash + MOD : hash), (hash2 < 0 ? hash2 + MOD2 : hash2)};
    }
};
```

## 8.3 Hash Range Query

```
struct Hash {
    const ll P = 31;
    const ll mod = 1e9 + 7;
    string s;
    int n;
    vll h, hi, p;
    Hash() {}
    Hash(string s) : s(s), n(s.size()), h(n), hi(n), p(n) {
        for (int i = 0; i < n; i++) p[i] = (i ? P * p[i - 1] : 1) % mod;
        for (int i = 0; i < n; i++) h[i] = (s[i] + (i ? h[i - 1] : 0) * P) % mod;
        for (int i = n - 1; i >= 0; i--)
            hi[i] = (s[i] + (i + 1 < n ? hi[i + 1] : 0) * P) % mod;
    }
    ll query(int l, int r) {
        ll hash = (h[r] - (l ? h[l - 1] * p[r - l + 1] % mod : 0));
        return hash < 0 ? hash + mod : hash;
    }
    ll query_inv(int l, int r) {
        ll hash = (hi[l] - (r + 1 < n ? hi[r + 1] * p[r - l + 1] % mod : 0));
        return hash < 0 ? hash + mod : hash;
    }
};
```

```
}
};
```

## 8.4 K-th digit in digit string

Find the k-th digit in a *digit string*, only works for  $1 \leq k \leq 10^{18}$  !  
Time: precompute  $O(1)$ , query  $O(1)$

```
using ull = vector<ull>;
ull pow10;
vector<array<ull, 4>> memo;
void precompute(int maxpow = 18) {
    ull qtd = 1;
    ull start = 1;
    ull end = 9;
    ull curlenght = 9;
    ull startstr = 1;
    ull endstr = 9;

    for (ull i = 0, j = 1ll; (int)i < maxpow; i++, j *= 10ll) pow10.eb(j);

    for (ull i = 0; i < maxpow - 1ull; i++) {
        memo.push_back({start, end, startstr, endstr});

        start = end + 1ll;
        end = end + (9ll * pow10[qtd]);
        curlenght = end - start + 1ull;

        qtd++;
        startstr = endstr + 1ull;
        endstr = (endstr + 1ull) + (curlenght)*qtd - 1ull;
    }
}
char kthDigit(ull k) {
    int qtd = 1;
    for (auto [s, e, ss, es] : memo) {
        if (k >= ss and k <= es) {
            ull pos = k - ss;
            ull index = pos / qtd;
            ull nmr = s + index;
            int i = k - ss - qtd * index;

            return ((nmr / pow10[qtd - i - 1]) % 10) + '0';
        }
        qtd++;
    }

    return 'X';
}
```

## 8.5 Longest Palindrome Substring (Manacher)

Finds the longest palindrome substring, manacher returns a vector where the i-th position is how much is possible to grow the string to the left and the right of i and keep it a palindrome.  
Time:  $O(N)$

```
vi manacher(string s) {
    string t2;
```

```

for (auto c : s) t2 += string("#") + c;
t2 = t2 + '#';
int n = t2.size();
t2 = "$" + t2 + "^";
vi p(n + 2);
int l = 1, r = 1;
for (int i = 1; i <= n; i++) {
    p[i] = max(0, min(r - i, p[l + (r - i)]));
    while (t2[i - p[i]] == t2[i + p[i]]) {
        p[i]++;
    }
    if (i + p[i] > r) {
        l = i - p[i], r = i + p[i];
    }
    p[i]--;
}
return vi(begin(p) + 1, end(p) - 1);
}

string longest_palindrome(const string &s) {
    vi xs = manacher(s);

    string s2;
    for (auto c : s) s2 += string("#") + c;
    s2 = s2 + '#';

    int mpos = 0;
    for (int i = 0; i < len(xs); i++) {
        if (xs[i] > xs[mpos]) {
            mpos = i;
        }
    }

    string ans;
    int k = xs[mpos];
    for (int i = mpos - k; i <= mpos + k; i++) {
        if (s2[i] != '#') {
            ans += s2[i];
        }
    }
    return ans;
}

void run() {
    string s;
    cin >> s;
    auto ans = longest_palindrome(s);
    cout << ans << endl;
}

```

## 8.6 Rabin Karp

```

size_t rabin_karp(const string &s, const string &p) {
    if (s.size() < p.size()) return 0;

    auto n = s.size(), m = p.size();
    const ll p1 = 31, p2 = 29, q1 = 1e9 + 7, q2 = 1e9 + 9;
    const ll p1_1 = fpow(p1, q1 - 2, q1), p1_2 = fpow(p1, m - 1, q1);
    const ll p2_1 = fpow(p2, q2 - 2, q2), p2_2 = fpow(p2, m - 1, q2);

```

```

pair<ll, ll> hs, hp;
for (int i = (int)m - 1; ~i; --i) {
    hs.first = (hs.first * p1) % q1;
    hs.first = (hs.first + (s[i] - 'a' + 1)) % q1;
    hs.second = (hs.second * p2) % q2;
    hs.second = (hs.second + (s[i] - 'a' + 1)) % q2;

    hp.first = (hp.first * p1) % q1;
    hp.first = (hp.first + (p[i] - 'a' + 1)) % q1;
    hp.second = (hp.second * p2) % q2;
    hp.second = (hp.second + (p[i] - 'a' + 1)) % q2;
}

size_t occ = 0;
for (size_t i = 0; i < n - m; i++) {
    occ += (hs == hp);

    int fi = s[i] - 'a' + 1;
    int fm = s[i + m] - 'a' + 1;

    hs.first = (hs.first - fi + q1) % q1;
    hs.first = (hs.first * p1_1) % q1;
    hs.first = (hs.first + fm * p1_2) % q1;
    hs.second = (hs.second - fi + q2) % q2;
    hs.second = (hs.second * p2_1) % q2;
    hs.second = (hs.second + fm * p2_2) % q2;
}
occ += hs == hp;

return occ;
}

```

## 8.7 String Psum

```

struct strPsum {
    ll n;
    ll k;
    vector<vll> psum;
    strPsum(const string &s) : n(s.size()), k(100), psum(k, vll(n + 1)) {
        for (ll i = 1; i <= n; ++i) {
            for (ll j = 0; j < k; ++j) {
                psum[j][i] = psum[j][i - 1];
            }
            psum[s[i - 1]][i]++;
        }
    }

    ll qtd(ll l, ll r, char c) { // [0,n-1]
        return psum[c][r + 1] - psum[c][l];
    }
}

```

## 8.8 Suffix Automaton (complete)

```

struct state {
    int len, link, cnt, firstpos;
    // this can be optimized using a vector with the alphabet size

```

```

    map<char, int> next;
    vi inv_link;
};

struct SuffixAutomaton {
    vector<state> st;
    int sz = 0;
    int last;
    vc cloned;

    SuffixAutomaton(const string &s, int maxlen)
        : st(maxlen * 2), cloned(maxlen * 2) {
        st[0].len = 0;
        st[0].link = -1;
        sz++;
        last = 0;
        for (auto &c : s) add_char(c);

        // precompute for count occurrences
        for (int i = 1; i < sz; i++) {
            st[i].cnt = !cloned[i];
        }
        vector<pair<state, int>> aux;
        for (int i = 0; i < sz; i++) {
            aux.push_back({st[i], i});
        }

        sort(all(aux), [](const pair<state, int> &a, const pair<state, int> &b) {
            return a.fst.len > b.fst.len;
        });

        for (auto &[stt, id] : aux) {
            if (stt.link != -1) {
                st[stt.link].cnt += st[id].cnt;
            }
        }

        // for find every occurende position
        for (int v = 1; v < sz; v++) {
            st[st[v].link].inv_link.push_back(v);
        }
    }

    void add_char(char c) {
        int cur = sz++;
        st[cur].len = st[last].len + 1;
        st[cur].firstpos = st[cur].len - 1;
        int p = last;
        // follow the suffix link until find a transition to c
        while (p != -1 and !st[p].next.count(c)) {
            st[p].next[c] = cur;
            p = st[p].link;
        }
        // there was no transition to c so create and leave
        if (p == -1) {
            st[cur].link = 0;
            last = cur;
            return;
        }
    }
};

```

```

    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
        st[cur].link = q;
    } else {
        int clone = sz++;
        cloned[clone] = true;
        st[clone].len = st[p].len + 1;
        st[clone].next = st[q].next;
        st[clone].link = st[q].link;
        st[clone].firstpos = st[q].firstpos;
        while (p != -1 and st[p].next[c] == q) {
            st[p].next[c] = clone;
            p = st[p].link;
        }
        st[q].link = st[cur].link = clone;
    }
    last = cur;
}

bool checkOccurrence(const string &t) { // O(len(t))
    int cur = 0;
    for (auto &c : t) {
        if (!st[cur].next.count(c)) return false;
        cur = st[cur].next[c];
    }
    return true;
}

ll totalSubstrings() { // distinct, O(len(s))
    ll tot = 0;
    for (int i = 1; i < sz; i++) {
        tot += st[i].len - st[st[i].link].len;
    }
    return tot;
}

// count occurrences of a given string t
int countOccurrences(const string &t) {
    int cur = 0;
    for (auto &c : t) {
        if (!st[cur].next.count(c)) return 0;
        cur = st[cur].next[c];
    }
    return st[cur].cnt;
}

// find the first index where t appears a substring O(len(t))
int firstOccurrence(const string &t) {
    int cur = 0;
    for (auto c : t) {
        if (!st[cur].next.count(c)) return -1;
        cur = st[cur].next[c];
    }
    return st[cur].firstpos - len(t) + 1;
}

vi everyOccurrence(const string &t) {
    int cur = 0;
}

```

```

for (auto c : t) {
    if (!st[cur].next.count(c)) return {};
    cur = st[cur].next[c];
}
vi ans;
getEveryOccurence(cur, len(t), ans);
return ans;
}

void getEveryOccurence(int v, int P_length, vi &ans) {
    if (!cloned[v]) ans.pb(st[v].firstpos - P_length + 1);
    for (int u : st[v].inv_link) getEveryOccurence(u, P_length, ans);
}
};

```

## 8.9 Z-function get occurence positions

$O(len(s) + len(p))$

```

vi getOccPos(string &s, string &p) {
    // Z-function
    char delim = '#';
    string t{p + delim + s};
    vi zs(len(t));

    int l = 0, r = 0;
    for (int i = 1; i < len(t); i++) {
        if (i <= r) zs[i] = min(zs[i - l], r - i + 1);
        while (zs[i] + i < len(t) and t[zs[i]] == t[i + zs[i]]) zs[i]++;
        if (r < i + zs[i] - 1) l = i, r = i + zs[i] - 1;
    }

    // Iterate over the results of Z-function to get ranges
    vi ans;
    int start = len(p) + 1 + 1 - 1;
    for (int i = start; i < len(zs); i++) {
        if (zs[i] == len(p)) {
            int l = i - start;
            ans.emplace_back(l);
        }
    }
    return ans;
}

```

## 9 Settings and macros

### 9.1 short-macro.cpp

```

#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define fastio \
    ios_base::sync_with_stdio(false); \
    cin.tie(0); \
    cout.tie(0);
#define len(__x) (int) __x.size()
using ll = long long;

```

```

using pii = pair<int, int>;
#define all(a) a.begin(), a.end()

void run() {}
int32_t main(void) {
    fastio;
    int t;
    t = 1;
    // cin >> t;
    while (t--) run();
}

```

### 9.2 .vimrc

```

set ts=4 sw=4 sta nu rnu sc cindent
set bg=dark ruler clipboard=unnamed,unnamedplus, timeoutlen=100
colorscheme default

```

```

nnoremap <C-j> :botright belowright term bash <CR>
syntax on

```

### 9.3 degug.cpp

```

#include <bits/stdc++.h>
using namespace std;
/***** Debug Code *****/
template <typename T>
concept Printable = requires(T t) {
    { std::cout << t } -> std::same_as<std::ostream &>;
};
template <Printable T>
void __print(const T &x) {
    cerr << x;
}
template <size_t T>
void __print(const bitset<T> &x) {
    cerr << x;
}
template <typename A, typename B>
void __print(const pair<A, B> &p);
template <typename... A>
void __print(const tuple<A...> &t);
template <typename T>
void __print(stack<T> s);
template <typename T>
void __print(queue<T> q);
template <typename T, typename... U>
void __print(priority_queue<T, U...> q);
template <typename A>
void __print(const A &x) {
    bool first = true;
    cerr << '{';
    for (const auto &i : x) {
        cerr << (first ? " " : ","), __print(i);
        first = false;
    }
    cerr << '>';
}

```



```

}
template <typename A, typename B>
void __print(const pair<A, B> &p) {
    cerr << '(';
    __print(p.first);
    cerr << ',';
    __print(p.second);
    cerr << ')';
}
template <typename... A>
void __print(const tuple<A...> &t) {
    bool first = true;
    cerr << '(';
    apply(
        [&first](const auto &...args) {
            ((cerr << (first ? "" : ","), __print(args), first = false), ...);
        },
        t);
    cerr << ')';
}
template <typename T>
void __print(stack<T> s) {
    vector<T> debugVector;
    while (!s.empty()) {
        T t = s.top();
        debugVector.push_back(t);
        s.pop();
    }
    reverse(debugVector.begin(), debugVector.end());
    __print(debugVector);
}
template <typename T>
void __print(queue<T> q) {
    vector<T> debugVector;
    while (!q.empty()) {
        T t = q.front();
        debugVector.push_back(t);
        q.pop();
    }
    __print(debugVector);
}
template <typename T, typename... U>
void __print(priority_queue<T, U...> q) {
    vector<T> debugVector;
    while (!q.empty()) {
        T t = q.top();
        debugVector.push_back(t);
        q.pop();
    }
    __print(debugVector);
}
void _print() { cerr << "]\n"; }
template <typename Head, typename... Tail>
void _print(const Head &H, const Tail &...T) {
    __print(H);
    if (sizeof...(T)) cerr << ", ";
    _print(T...);
}

```

```

#define dbg(x...) \
    cerr << "[" << #x << "]" = ["; \
    _print(x)

```

## 9.4 .bashrc

```

cpp() {
    echo ">> COMPILING <<" 1>&2
    g++ -std=c++17 \
        -O2 \
        -g \
        -g3 \
        -Wextra \
        -Wshadow \
        -Wformat=2 \
        -Wconversion \
        -fsanitize=address,undefined \
        -fno-sanitize-recover \
        -Wfatal-errors \
        $1

    if [ $? -ne 0 ]; then
        echo ">> FAILED <<" 1>&2
        return 1
    fi
    echo ">> DONE << " 1>&2
    time ./a.out ${@:2}
}

prepare() {
    for i in {a..z}
    do
        cp macro.cpp $i.cpp
        touch $i.py
    done

    for i in {1..10}
    do
        touch in${i}
        touch out${i}
        touch ans${i}
    done
}

```

## 9.5 macro.cpp

```

#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define fastio \
    ios_base::sync_with_stdio(false); \
    cin.tie(0); \
    cout.tie(0);
#define len(__x) (int) __x.size()
using ll = long long;
using ull = unsigned long long;
using ld = long double;

```

```

using vll = vector<ll>;
using pll = pair<ll, ll>;
using vll2d = vector<vll>;
using vi = vector<int>;
using vi2d = vector<vi>;
using pii = pair<int, int>;
using vii = vector<pii>;
using vc = vector<char>;
#define all(a) a.begin(), a.end()
#define pb(____x) push_back(____x)
#define mp(____a, ____b) make_pair(____a, ____b)
#define eb(____x) emplace_back(____x)

```

```

vector<string> dir({"LU", "U", "RU", "R", "RD", "D", "LD", "L"});
int dx[] = {-1, -1, -1, 0, 1, 1, 1, 0};
int dy[] = {-1, 0, 1, 1, 1, 0, -1, -1};
const ll oo = 1e18;

auto solve() {}
int32_t main(void) {
    fastio;
    int t;
    t = 1;
    // cin >> t;
    while (t--) solve();
}

```