Universidade de Brasilia

Tomate Cereja

Wallace Wu, Pedro Gallo, Lucas Sala

2024-08-14

Contest (1)

template.cpp

44 lines

```cpp
// #pragma GCC optimize("O3,unroll-loops")
// #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
#include <bits/stdc++.h>
using namespace std;
#define sws cin.tie(0)->sync_with_stdio(0)

typedef long long ll;
typedef long double ld;
#define endl '\n'
#define pb push_back
#define ff first
#define ss second
#define pll pair<ll, ll>
#define vll vector<ll>

#define teto(a, b) (((a)+(b)-1)/(b))
#define LSB(i) ((i) & -(i))
#define MSB(i) (63 - __builtin_clzll(i))
#define BITS(i) __builtin_popcountll(i)

template<class A, class B> auto& operator<<(ostream &os, pair<A
    , B> p) {
    return os << '{' << p.ff << ", " << p.ss << '}'; }
template<class... A>void db(A ...a) {((cerr<<"{"<<a<<"} "),
    ...); cerr<<endl;}
template<class A>void db2(A a) {for(auto b:a) {cerr<<b<<" ";}
    cerr<<endl;}
template<class A>void db3(A a) {for(auto b:a) db2(b);}
#define dbg(...) {cerr << #__VA_ARGS__ << ": "; db(__VA_ARGS__)
    ; }
#define deb(a) {cerr << #a << ": "; db2(a); }
#define debug(a) {cerr << #a << ": " << endl; db3(a); }

const ll MAX = 2e5+10;
const ll MOD = 998'244'353;
const ll INF = 1e18; // INT32_MAX
const ld EPS = 1e-7;
const ld PI = acos(-1);

#include <chrono>
using namespace std::chrono;

int32_t main(){ sws;
    auto start = high_resolution_clock::now();
    // function to be timed here
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
    cout << duration.count() << endl;
}
```

.bashrc

1 lines

```
alias comp='g++ -std=c++17 -O2 -g3 -ggdb3 -fsanitize=address,
    undefined -Wall -Wextra -Wshadow -Wconversion -o test'
```

troubleshoot.txt

52 lines

```
Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?
```

Data structures (2)

2.1 Stack

An optimization for *std::stack* is to use a *std::vector* as the container, instead of *std::deque* !

```cpp
stack<int, vector<int>> st;
```

A stack can be used to efficiently solve the maximum rectangle in a histogram problem:

max-rectangle-histogram.cpp

Description: solves the problem of finding the maximum rectangle area in a grid setting (different widths, different heights)

Time: $\mathcal{O}(nm)$

461370, 54 lines

```cpp
// Example Problem: You are given a map of a forest where some
    squares are empty and some squares have trees.
// What is the maximum area of a rectangular building that can
    be placed in the forest so that no trees must be cut down?

ll maxRectangleHistogram(vector<ll> x) { // O(n)

    // add an end point with heigth 0 to compute the last
        rectangles
    x.pb(0);

    ll area = 0;
    ll n = x.size();
    stack<pll, vector<pll>> st; // {maxLeft, height for this
        rectangle}

    for(ll i=0; i<n; i++) {
        ll h = x[i];
        ll maxLeft = i;

        while(!st.empty() and st.top().ss >= h) {
            auto [maxLeft2, h2] = st.top(); st.pop();

            // compute the area of the de-stacked rectangle
            area = max(area, (i-maxLeft2)*h2 );

            // extend current rectangle width with previous
            maxLeft = maxLeft2;
        }

        st.push({maxLeft, h});
    }

    return area;
}

int32_t main(){ sws;
    ll n, m; cin >> n >> m;

    vector<vector<ll>> grid(n, vector<ll>(m));

    // convert the problem into N histogram subproblems, O(n m)
    for(ll i=0; i<n; i++) {
        for(ll j=0; j<m; j++) {
            char c; cin >> c;
            if (c == '*') grid[i][j] = 0;
            else if (i == 0) grid[i][j] = 1;
            else grid[i][j] = grid[i-1][j] + 1;
        }
    }

    ll area = 0;
```

```
    for(ll i=0; i<n; i++) {
        area = max(area, maxRectangleHistogram(grid[i]));
    }

    cout << area << endl;
}
```

Also can be used to solve the maximum rectangle in a grid, with some blocked spots:

max-rectangle-grid.cpp
Description: solves the problem of finding the maximum rectangle area in a histogram setting (same bottom, different heights).
Time: $\mathcal{O}(n)$
8610da, 38 lines

```
// Example Problem: A fence consists of n vertical boards. The
     width of each board is 1 and their heights may vary.
// You want to attach a rectangular advertisement to the fence.
     What is the maximum area of such an advertisement?

ll maxRectangleHistogram(vector<ll> x) { // O(n)

    // add an end point with heigth 0 to compute the last
         rectangles
    x.pb(0);

    ll area = 0;
    ll n = x.size();
    stack<pll, vector<pll>> st; // {maxLeft, height for this
         rectangle}

    for(ll i=0; i<n; i++) {
        ll h = x[i];
        ll maxLeft = i;

        while(!st.empty() and st.top().ss >= h) {
            auto [maxLeft2, h2] = st.top(); st.pop();

            // compute the area of the de-stacked rectangle
            area = max(area, (i-maxLeft2)*h2 );

            // extend current rectangle width with previous
            maxLeft = maxLeft2;
        }

        st.push({maxLeft, h});
    }

    return area;
}

int32_t main(){ sws;
    ll n; cin >> n;
    vector<ll> x;
    for(ll i=0, a; i<n; i++) cin >> a, x.pb(a);
    cout << maxRectangleHistogram(x) << endl;
}
```

2.2 List

std::list is a container that supports constant time insertion and removal of elements from anywhere in the container.

Adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references. An iterator is invalidated only when the corresponding element is deleted.

Element Access: O(1)

- *list.back()*
- *list.front()*

Modifiers: O(1)

- *list.insert(itr, val)* inserts val before itr and returns an itr to the inserted value
- *list.erase(itr)* erases the element referenced by itr and returns the itr for the next value (or .end())
- *list.push_back(val)*
- *list.pop_back(val)*
- *list.push_front(val)*
- *list.pop_back(val)*

2.3 Ordered Set

Policy Based Data Structures (PBDS) from gcc compiler

Ordered Multiset can be created using ordered_set<pll>val, idx

order_of_key() can search for non-existent keys!

find_by_order() requires existent key and return the 0-idx position of the given value. Therefore, it returns the numbers of elements that are smaller than the given value;

ordered-set.cpp
Description: Set with index operators, implemented by gnu pbds. Remember to compile with gcc!!
Time: $\mathcal{O}(\log(N))$ but with slow constant
<bits/extc++.h>, <bits/extc++.h> 8578e5, 11 lines

```
// 0-idx
// find_by_order(i) -> iterator to elem with index i
// order_of_key(val) -> index of key

// Ordered Set
using namespace __gnu_pbds;
template <class T> using ordered_set = tree<T, null_type, less<
    T>, rb_tree_tag, tree_order_statistics_node_update>;

// Ordered Map
using namespace __gnu_pbds;
template <class K, class V> using ordered_map = tree<K, V, less
    <K>, rb_tree_tag, tree_order_statistics_node_update>;
```

2.3.1 Pyramid Array min-cost

You are given an array consisting of n integers. On each move, you can swap any two adjacent values. You want to transform the array into a pyramid array. This means that the final array has to be first increasing and then decreasing. It is also allowed that the final array is only increasing or decreasing. What is the minimum number of moves needed?

pyramid-array.cpp
Description: algorithm to find the min-cost of sorting an array in a pyramid order
Time: $\mathcal{O}(N \log(N))$, or $\mathcal{O}\left(N \log^2(N)\right)$ if iterating the map directly
165fd7, 25 lines

```
int32_t main() { sws;
    ll n; cin >> n;
    map<ll, vll> freq;
    for(ll i=0; i<n; i++) {
        ll val; cin >> val;
        freq[val].pb(i);
    }
    ordered_set<ll> os; // os with indexes of greater processed
         elements
    ll ans = 0;
    // iterate from greater values to lesser one.
    // for each element,
    // consider inserting it to the left of all greater
         elements
    // or to the right of all greater elements
    for(auto itr = freq.rbegin(); itr != freq.rend(); itr++) {
        auto [val, vec] = *itr;
        for(auto idx : vec) {
            ll pos = os.order_of_key({idx});
            ll left_cost = pos;
            ll right_cost = (ll)os.size() - pos;
            ans += min(left_cost, right_cost);
        }
        for(auto idx : vec) os.insert(idx);
    }
    cout << ans << endl;
}
```

2.4 Interval Set

interval-set.cpp
Description: A set that contains closed [l, r] interval which are disjoint (no intersection). This set is ordered and each interval [l1, r1] < [l2, r2] has r1 < l2. When a new interval is added, it checks which intersections will occur and rearranges the intervals.
Time: $\mathcal{O}(\log(N))$ per insertion, slow constant
a3c7e0, 29 lines

```
// keeps track of disjoint closed intervals [l, r]
// a new interval added may replace parts of an older one
struct IntervalSet {
    using T = array<ll, 3>;
    set<T> ranges;

    void add(T arr) {
        auto [l, r, k] = arr;

        while(ranges.upper_bound({r, INF, INF}) != ranges.begin
            ()) {
            auto itr = prev(ranges.upper_bound({r, INF, INF}));
            auto [l2, r2, k2] = *itr;

            if (r2 < l) break;
            // garantees that there is an intersection: l2 <= r
                 and r2 >= l

            ranges.erase(itr);

            if (l2 <= l-1) {
                ranges.insert({l2, l-1, k2});
            }

            if (r+1 <= r2) {
                ranges.insert({r+1, r2, k2});
            }
        }
```

```
        ranges.insert({l, r, k});
    }
};
```

2.5 Disjoint Set Union

There are two optional improvements:

- Tree Balancing
- Path Compression

If one improvement is used, the time complexity will become $O(\log N)$

If both are used, $O(\alpha) \approx O(5)$

In addition, the **rollback operation** may be implemented, but it requires to exclude *path compression* optimization.

dsu.cpp
Description: Disjoint Set Union with path compression and tree balancing
Time: $\mathcal{O}(\alpha)$
467ae9, 21 lines

```
struct DSU {
    vector<ll> group, card;

    DSU (ll n) : group(n+1), card(n+1, 1) { // 1-idx
        iota(group.begin(), group.end(), 0);
    }

    ll find(ll i) {
        return (i == group[i]) ? i : (group[i] = find(group[i])
            );
    }

    // returns false if a and b are already in the same
        component
    bool join(ll a, ll b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (card[a] < card[b]) swap(a, b);
        card[a] += card[b];
        group[b] = a;
        return true;
    }
};
```

dsu-rollback.cpp
Description: Disjoint Set Union with snapshot, tree balancing and WITHOUT path compression
Time: $\mathcal{O}(\log n)$
8138ce, 40 lines

```
// with rollback and numbers of comps
// without path compression, therefore O(log n)
struct DSU {
    vector<ll> group, card;
    vector<pair<ll &, ll>> history;
    ll comps;

    DSU (ll n) : group(n+1), card(n+1, 1) { // 1-idx
        iota(group.begin(), group.end(), 0);
        comps = n; // don't include 0
    }

    ll find(ll i){
        return (i == group[i]) ? i : find(group[i]);
    }

    void join(ll a ,ll b){
        a = find(a), b = find(b);
        if (a == b) return;

        if (card[a] < card[b]) swap(a, b);

        history.pb({card[a], card[a]});
        history.pb({group[b], group[b]});
        history.pb({comps, comps});

        comps -= 1;
        card[a] += card[b];
        group[b] = a;
    }

    ll snapshot() { return history.size(); }

    void rollback(ll until) { // restore to snapshot == until
        while(snapshot() > until) {
            history.back().ff = history.back().ss;
            history.pop_back();
        }
    }
};
```

2.5.1 Dynamic Connectivity

Consider an undirected graph that consists of n nodes and m edges. There are two types of events that can happen:

- A new edge is created between nodes a and b.
- An existing edge between nodes a and b is removed.

Your task is to report the number of components after every event (and before all events).

query-tree.cpp
Description: All queries have an active intervals, build a tree to store these queries and iterate it in dfs order with rollbacks. The code below solves the specific problem of Dynamic Conectivity.
Time: $\mathcal{O}(n \log^2(n))$
32b252, 97 lines

```
// include struct DSU {} (with rollback)

ll L=0, R;
struct QueryTree {
    struct Query {
        ll l, r; // this ranges is active in [l, r]
        ll u, v; // edge {u, v} will be merged in DSU
    };

    // each node is a vector of queries
    vector<vector<Query>> tree;

    QueryTree(ll n) {
        R = n;
        tree.assign(4*n + 10, {});
    }

    // l, r (tree); left, right (query)
    void add(Query q, ll l=L, ll r=R, ll i=1) {
        auto [left, right] = tie(q.l, q.r);

        if (right < l or r < left) return;

        if (left <= l and r <= right) {
            tree[i].pb(q);
            return;
        }

        ll mid = (l+r)/2;
        add(q, l, mid, 2*i);
        add(q, mid+1, r, 2*i+1);
    }

    void dfs(DSU &dsu, vector<ll> &ans, ll i = 1, ll l=L, ll r=
        R) {
        ll snap = dsu.snapshot();

        for(auto &q : tree[i]) {
            dsu.join(q.u, q.v);
        }

        if (l == r) { // leaf
            ans[l] = dsu.comps;
        }
        else {
            ll mid = (l + r)/2;
            dfs(dsu, ans, 2*i, l, mid);
            dfs(dsu, ans, 2*i + 1, mid+1, r);
        }

        // rollback
        dsu.rollback(snap);
    }
};

int32_t main(){ sws;
    ll n, m, k; cin >> n >> m >> k;

    QueryTree tree(k);
    map<pll, pll> queries;

    for(ll i=0; i<m; i++) { // time = 0
        ll u, v; cin >> u >> v;
        if (u > v) swap(u, v);
        queries[{u, v}] = {0, k};
    }

    for(ll t=1; t<=k; t++) {
        ll op, u, v; cin >> op >> u >> v;
        if (u > v) swap(u, v);
        if (op == 1) {
            queries[{u, v}] = {t, k};
        }
        else {
            queries[{u, v}].ss = t-1;
            QueryTree::Query q;
            tie(q.l, q.r) = queries[{u, v}];
            tie(q.u, q.v) = {u, v};
            tree.add(q);
            queries.erase({u, v});
        }
    }

    for(auto [key, range] : queries) {
        QueryTree::Query q;
        tie(q.l, q.r) = range;
        tie(q.u, q.v) = key;
        tree.add(q);
    }

    vector<ll> ans(k+1);
    DSU dsu(n);
    tree.dfs(dsu, ans);

    for(auto val : ans)
```

```
        cout << val << " ";
    cout << endl;
}
```

2.6 Trie

Also called a **digital tree** or **prefix tree**.

trie.cpp
Description: Creates a trie by pre-allocating the trie array, which contains the indices for the child nodes. The trie can be easily modified to support alphanumeric strings instead of binary strings.
Time: $\mathcal{O}(D)$, D = depth of trie
86b236, 21 lines

```
ll alphabet = 26;

struct Trie {
    vector<vll> t;
    vector<ll> words;
    ll idx = 1;

    // n = maximum number of nodes to be created = N*len
    Trie(ll n) : t(n, vll(alphabet, 0)), words(n, 0) {}

    void add(string s){ // O(Depth)
        ll node = 0;
        for(auto c : s) {
            if(t[node][c-'a'] == 0) { // create new node
                t[node][c-'a'] = idx++;
            }
            node = t[node][c-'a'];
        }
        words[node]++;
    }
};
```

2.7 Sparse Table

There are two requisites for using sparse tables:

- **Immutability** on the array.
- **Associative** property on the operator.

And for the $O(1)$ complexity, the operator also has to have the **Idempotence** property (overlap friendly/agnostic). Meaning that when considering $f([a,c], f[b,d])$, it has the same value as $f([a,d])$ even though there is an overlap in $[b,c]$.
$(a \leq b \leq c \leq d)$.

Associative Operators:

sum, product, xor, concatenation, union

Associative and Idempotent Operators:

min, max, gcd, lcm, and, or, intersection

sparse-table.cpp
Description: By precomputing for each position and each power of two the value of a range, Anwer quickly any query.
Time: $\mathcal{O}(n \log n)$ for precomputing, $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ per query
c21bf3, 52 lines

```
// computes the MSB = the floor of log2(i) in O(1)
// MSB(0) = -1
#define MSB(i) (64 - 1 - __builtin_clzll(i))

template<class T>
struct SparseTable {

    // the function must be associative !!!
    T f(T a, T b) {
        return min(a, b);
    }

    ll n, m;
    vector<vector<T>> st; // st[j][i] covers range [i, i+2^j)

    // 0-idx: [0, n)
    SparseTable(vector<T> &v) : n(v.size()), st({v}) {
        m = MSB(n);

        for(ll k=1; k<=m; k++) {
            st.emplace_back(n);
            for(ll i=0; i + (1LL << k) <= n; i++) {
                st[k][i] = f(
                    st[k-1][i],
                    st[k-1][i + (1LL << (k-1))]
                );
            }
        }
    }

    // constant query for functions with Idempotence (overlap friendly)
    T query(ll l, ll r) { // query for [l, r] in O(1)
        if (l == r) return st[0][l];
        ll k = MSB(r-l+1);
        return f(
            st[k][l],
            st[k][r - (1LL << k) + 1]
        );
    }

    // logarithmic query for functions without Idempotence
    T query(ll l, ll r) { // query for [l, r] in O(log)
        T ans = {}; // define the correct default null value here
        for(ll k=m; k>=0; k--) {
            if ((1LL << k) <= (r-l+1)) {
                ans = f(ans, st[k][l]);
                l += 1LL << k;
            }
        }
        return ans;
    }
};
```

2.8 Fenwick Tree

Also called **Binary Indexed Tree (BIT)**.

Observation: BIT cannot support min/max queries, because it's mandatory to have an inverse operation.

Let's define $g(i)$ as the number acquired after removing the $LSB(i)$ from i:

$$g(i) = i - LSB(i) = i - (i\&(-i))$$

Then, each value of the **Bit vector** will be resposible to store the range value of the interval:

$$(g(i), i]$$

Therefore, to retrieve the value in an arbitrary range $[1, x]$, it's only necessary to merge:

$$Bit[i] + Bit[g(i)] + Bit[g(g(i))] + ... + Bit[last], last >= 1$$

In the **1-Indexed implementation**, $Bit[0]$ is undefined and not used.

fenwick-tree.cpp
Description: Simple 1D Fenwick Tree with point increase, range sum query. 1-idx
Time: $\mathcal{O}(\log(n))$ to add, get psum, or range sum query
f6b1d5, 26 lines

```
// 1-idx, vector covers [1, n]
struct FT {
    ll n;
    vector<ll> bit;

    FT(ll sz) : n(sz), bit(sz+1, 0) { }

    // add delta to positon pos
    void add(ll pos, ll delta) { // O(log(n))
        for (; pos <= n; pos += pos & -pos)
            bit[pos] += delta;
    }

    // get prefix sum of [1, pos]
    ll sum(ll pos) { // O(log(n))
        ll ans = 0;
        for (; pos >= 1; pos -= pos & -pos)
            ans += bit[pos];
        return ans;
    }

    // query the sum of range [l, r]
    ll query(ll l, ll r) { // O(log(n))
        return sum(r) - sum(l - 1);
    }
};
```

fenwick-tree-2D.cpp
Description: Simple 2D Fenwick Tree with point increase, 2D range sum query. 1-idx
Time: $\mathcal{O}(\log(n) \cdot \log(m))$ to add, get psum, or range sum query
9e6e9a, 28 lines

```
// 1-idx, cover the grid of rows [1, n] and columns [1, m]
struct FT2D {
    ll n, m;
    vector<vll> bit;

    FT2D(ll nn, ll mm) : n(nn), m(mm) {
        bit.assign(n+1, vll(m+1, 0));
    }

    void add(ll x, ll y, ll delta) { // O(log(n)*log(m))
        for(ll i=x; i<=n; i += i & -i)
            for(ll j=y; j<=m; j += j & -j)
                bit[i][j] += delta;
    }

    ll sum(ll x, ll y) { // O(log(n)*log(m))
        ll ans = 0;
```

```
        for(ll i=x; i>=1; i -= i & -i)
            for(ll j=y; j>=1; j -= j & -j)
                ans += bit[i][j];
        return ans;
    }

    ll query(ll x1, ll y1, ll x2, ll y2) { // O(log(n)*log(m))
        x1--; y1--; // to make point {x1, y1} inclusive
        return sum(x2, y2) - sum(x2, y1) - sum(x1, y2) + sum(x1
            , y1);
    }
};
```

2.9 Segment Trees

Each node of the segment tree represents the cumulative value of a range.

Observation: For some problems, such as range distinct values query, considerer offiline approach, ordering the queries by L for example.

Implementation Observation:

- *left/right* are the range limits for the query;
- *l/r* are the internal range variables of the tree;

2.9.1 Recursive SegTree

seg-recursive-sum.cpp

Description: Basic Recursive Segment Tree for points increase and range sum query. When initializing, choose an appropriate value for n = R and call build()

Time: $\mathcal{O}(N \log(N))$ to build, $\mathcal{O}(\log(N))$ to increase or query b1e81c, 69 lines

```
// [0, n] segtree for range sum query, point increase
struct Segtree {
    struct Node {
        // correctly initialize default null values:
        ll sum = 0;
    };

    ll L=0, R;
    vector<ll> v;
    vector<Node> t; // tree

    Segtree(ll n) : R(n), v(n+1), t(4*(n+1)) {}

    Node merge(Node a, Node b) {
        return Node {
            // merge operation:
            a.sum + b.sum
        };
    }

    void build(ll l, ll r, ll i=1) {
        if (l == r) {
            t[i] = Node {
                // leaf element:
                v[l]
            };
            return;
        }
        ll mid = (l+r)/2;
        build(l, mid, 2*i);
        build(mid+1, r, 2*i+1);
        t[i] = merge(t[2*i], t[2*i+1]);
    }
    void build() {
        build(L, R, 1);
    }

    void increase(ll pos, ll inc, ll l, ll r, ll i) {
        if (l == r) {
            // increase operation:
            t[i].sum += inc;
            return;
        }
        ll mid = (l+r)/2;
        if (pos <= mid)
            increase(pos, inc, l, mid, 2*i);
        else
            increase(pos, inc, mid+1, r, 2*i+1);
        t[i] = merge(t[2*i], t[2*i+1]);
    }
    void increase(ll pos, ll inc) {
        increase(pos, inc, L, R, 1);
    }

    // [a, b] are the range limits for the query
    // [l, r] are the internal variables of the t
    Node query(ll a, ll b, ll l, ll r, ll i) {
        if (b < l or r < a) return Node{}; // default null
            value
        else if (a <= l and r <= b) return t[i];
        ll mid = (l+r)/2;
        return merge(
            query(a, b, l, mid, 2*i),
            query(a, b, mid+1, r, 2*i+1)
        );
    }
    Node query(ll a, ll b) {
        return query(a, b, L, R, 1);
    }
};
```

seg-recursive-minmax.cpp

Description: Basic Recursive Segment tree for point update, range min/-max query When initializing, choose an appropriate value for (n = R) and call build()

Time: $\mathcal{O}(N \log(N))$ to build, $\mathcal{O}(\log(N))$ to update or query 8cb53c, 71 lines

```
// [0, n] segtree for point assignment update, range min/max
    query
struct Segtree {
    struct Node {
        // correctly initialize default null values:
        ll mn = INF, mx = -INF;
    };

    ll L=0, R;
    vector<ll> v;
    vector<Node> t;

    Segtree(ll n) : R(n), v(n+1), t(4*(n+1)) {}

    Node merge(Node a, Node b) {
        return Node {
            // merge operation:
            min(a.mn, b.mn),
            max(a.mx, b.mx)
        };
    }

    void build(ll l, ll r, ll i) {
        if (l == r) {
            t[i] = Node {
                // leaf element:
                v[l],
                v[l]
            };
            return;
        }
        ll mid = (l+r)/2;
        build(l, mid, 2*i);
        build(mid+1, r, 2*i+1);
        t[i] = merge(t[2*i], t[2*i+1]);
    }
    void build() {
        build(L, R, 1);
    }

    void update(ll pos, ll val, ll l, ll r, ll i) {
        if (l == r) {
            // update(assignment) operation:
            t[i].mn = t[i].mx = val;
            return;
        }
        ll mid = (l+r)/2;
        if (pos <= mid)
            update(pos, val, l, mid, 2*i);
        else
            update(pos, val, mid+1, r, 2*i+1);
        t[i] = merge(t[2*i], t[2*i+1]);
    }
    void update(ll pos, ll val) {
        update(pos, val, L, R, 1);
    }

    // [a, b] are the range limits for the query
    // [l, r] are the internal variables of the t
    Node query(ll a, ll b, ll l, ll r, ll i) {
        if (b < l or r < a) return Node{}; // default null
            value
        else if (a <= l and r <= b) return t[i];
        ll mid = (l+r)/2;
        return merge(
            query(a, b, l, mid, 2*i),
            query(a, b, mid+1, r, 2*i+1)
        );
    }
    Node query(ll a, ll b) {
        return query(a, b, L, R, 1);
    }
};
```

s

2.9.2 Inverted Segtree

Instead of keeping the prefix sum for all the children in each node, store only the delta encoding value.

Therefore, to check a value in a certain position, iterate and sum all delta values from root to leaf.

seg-inverted.cpp

Description: Basic Inverted Segment tree for point query stored value, range increase When initializing, choose an appropriate value for n=R.

Time: $\mathcal{O}(N \log(N))$ to build, $\mathcal{O}(\log N)$ to range increase or point query 43e32f, 66 lines

```
// [0, n] segtree for point query stored value, range increase
struct Segtree {
    struct Node {
        // correctly initialize default null values:
```

```
        ll sum = 0;
    };

    ll L=0, R;
    vector<ll> v;
    vector<Node> t;

    Segtree(ll n) : R(n), v(n+1), t(4*(n+1)) {}

    Node merge(Node a, Node b) {
        return Node {
            // merge operation:
            a.sum + b.sum
        };
    }

    void build(ll l, ll r, ll i) {
        if (l == r) {
            t[i] = Node {
                // leaf element:
                v[l]
            };
            return;
        }
        ll mid = (l+r)/2;
        build(l, mid, 2*i);
        build(mid+1, r, 2*i+1);
        t[i] = Node{};
    }
    void build() {
        build(L, R, 1);
    }

    // [a, b] are the range limits for the query
    // [l, r] are the internal variables of the t
    void increase(ll inc, ll a, ll b, ll l, ll r, ll i) {
        if (b < l or r < a) return;
        else if (a <= l and r <= b) {
            // increase operation
            t[i].sum += inc;
            return;
        }
        ll mid = (l+r)/2;
        increase(inc, a, b, l, mid, 2*i);
        increase(inc, a, b, mid+1, r, 2*i+1);
    }
    void increase(ll inc, ll a, ll b) {
        increase(inc, a, b, L, R, 1);
    }

    Node query(ll pos, ll l, ll r, ll i) {
        if (l == r) return t[i];
        ll mid = (l+r)/2;
        if (pos <= mid)
            return merge(t[i], query(pos, l, mid, 2*i));
        else
            return merge(t[i], query(pos, mid+1, r, 2*i+1));
    }
    Node query(ll pos) {
        return query(pos, L, R, 1);
    }
};
```

2.9.3 Lazy Segtree

seg-lazy-sum.cpp

Description: Segtree with range sum query, range assignment update, range increase update.

Time: $\mathcal{O}(\log(n))$ for each query, $\mathcal{O}(n \log(n))$ to build() 086cae, 103 lines

```
// [0, n] segtree for range sum query, range assignment update,
     range increase update.
struct SegtreeLazy {
    struct Node {
        // correctly initialize default null values:
        ll sum = 0;
    };

    ll L=0, R;
    vector<ll> v;
    vector<Node> t;
    vector<pll> lazy; // {val, type}
    // type = 0 (no pending update); type = 1 (increase); type
         = 2 (assign)

    SegtreeLazy(ll n) : R(n), v(n+1), t(4*(n+1)), lazy(4*(n+1))
         {}

    Node merge(Node a, Node b) {
        return Node{
            a.sum + b.sum
        };
    }

    void build(ll l, ll r, ll i) {
        if (l == r) {
            t[i] = Node{
                v[l]
            };
        }
        else {
            ll mid = (l+r)/2;
            build(l, mid, 2*i);
            build(mid+1, r, 2*i+1);
            t[i] = merge(t[2*i], t[2*i+1]);
        }
        lazy[i] = {0, 0};
    }
    void build() {
        build(L, R, 1);
    }

    void push(ll l, ll r, ll i) {
        if (lazy[i].ss) {
            auto [val, type] = lazy[i];
            if (type == 1) { // increase type
                t[i].sum += val * (r-l+1);
            }
            else if (type == 2) { // assignment type
                t[i].sum = val * (r-l+1);
            }
            if (l != r) { // push downwards
                for(auto j : {2*i, 2*i+1}) {
                    auto &[val2, type2] = lazy[j];
                    if (type2 == 0) { // child is empty => just
                         replace
                        lazy[j] = lazy[i];
                    }
                    else if (type == type2) { // same type =>
                         just push
                        if (type == 1) val2 += val;
                        else val2 = val;
                    }
                    else if (type2 == 1 and type == 2) { //
                         increase then assign
                        lazy[j] = lazy[i]; // just ignore past
                             increasing and only assign
                    }
                    else if (type2 == 2 and type == 1) { //
                         assign then increase
                        val2 += val; // keep the assignment,
                             but increase it
                    }
                }
            }
            lazy[i] = {0, 0};
        }
    }

    // [a, b] are the range limits for the query
    // [l, r] are the internal variables of the t
    void update(ll val, ll type, ll a, ll b, ll l, ll r, ll i)
         {
        push(l, r, i);
        if (b < l or r < a) return;
        else if (a <= l and r <= b){
            lazy[i] = {val, type};
            push(l, r, i);
            return;
        }
        ll mid = (l+r)/2;
        update(val, type, a, b, l, mid, 2*i);
        update(val, type, a, b, mid+1, r, 2*i+1);
        t[i] = merge(t[2*i], t[2*i+1]);
    }
    void update(ll val, ll type, ll a, ll b) {
        update(val, type, a, b, L, R, 1);
    }

    Node query(ll a, ll b, ll l, ll r, ll i) {
        push(l, r, i);
        if (b < l or r < a) return Node{}; // default null
             value
        else if (a <= l and r <= b) return t[i];
        ll mid = (l+r)/2;
        return merge(
            query(a, b, l, mid, 2*i),
            query(a, b, mid+1, r, 2*i+1)
        );
    }
    Node query(ll a, ll b) {
        return query(a, b, L, R, 1);
    }
};
```

2.9.4 PA Segtree

seg-pa.cpp

Description: Seg with PA (Progressao Aritmetica / Arithmetic Progression) When initializing the segmente tree, remeber to choose a proper value for (n=R) and call build()

Time: $\mathcal{O}(N \log(N))$ to build, $\mathcal{O}(\log(N))$ to increase or query 43d355, 95 lines

```
inline pll operator +(pll a, pll b) {
    return {a.ff + b.ff, a.ss + b.ss};
}

// [0, n] segtree for range sum query, range increase with PA
struct SegtreePA {
    struct Node {
        // correctly initialize default null values:
        ll sum = 0;
    };

    ll L=0, R;
    vector<ll> v;
    vector<Node> t;
    vector<pll> lazy; // {x, y} of the expression: x*i + y
```

```
// PA: x = ratio, y = constant

SegtreePA(ll n) : R(n), v(n+1), t(4*(n+1)), lazy(4*(n+1)) {
    }

Node merge(Node a, Node b) {
    return Node {
        // merge operaton:
        a.sum + b.sum
    };
}

void build(ll l, ll r, ll i) {
    if (l == r) {
        t[i] = Node {
            // leaf element:
            v[l]
        };
    }
    else {
        ll mid = (l+r)/2;
        build(l, mid, 2*i);
        build(mid+1, r, 2*i+1);
        t[i] = merge(t[2*i], t[2*i+1]);
    }
    lazy[i] = {0, 0};
}
void build() {
    build(L, R, 1);
}

void push(ll l, ll r, ll i) {
    auto [x, y] = lazy[i];
    if (x == 0 and y == 0) return;
    ll len = r-l+1;
    // (l_val + r_val) * len / 2
    Node val{ ((y + y + x*(len-1))*len) / 2 };
    t[i] = merge(t[i], val);
    if (l != r) {
        ll mid = (l+r)/2;
        lazy[2*i] = lazy[2*i] + lazy[i];
        lazy[2*i+1] = lazy[2*i+1] + pll{x, y + x*(mid-l+1)}
            ;
    }
    lazy[i] = {0, 0};
}

void increase(ll x, ll y, ll a, ll b, ll l, ll r, ll i) {
    // PA: x = ratio, y = constant
    // [a, b] are the range limits for the query
    // [l, r] are the internal variables of the t
    push(l, r, i);
    if (b < l or r < a) return;
    else if (a <= l and r <= b) {
        lazy[i] = {x, y};
        push(l, r, i);
        return;
    }
    ll mid = (l+r)/2;
    increase(x, y, a, b, l, mid, 2*i);
    ll ny = y + max( x*( mid-max(a, l) + 1), 0LL);
    increase(x, ny, a, b, mid+1, r, 2*i+1);
    t[i] = merge(t[2*i], t[2*i+1]);
}
void increase(ll x, ll y, ll a, ll b) {
    increase(x, y, a, b, L, R, 1);
}

Node query(ll a, ll b, ll l, ll r, ll i) {
    push(l, r, i);
    if (b < l or r < a) return Node{}; // default null
        value
    else if (a <= l and r <= b) return t[i];
    ll mid = (l+r)/2;
    return merge(
        query(a, b, l, mid, 2*i),
        query(a, b, mid+1, r, 2*i+1)
    );
}
Node query(ll a, ll b) {
    return query(a, b, L, R, 1);
}
};
```

2.10 Treap

treap.cpp
Description: Implicit Treap
Time: $\mathcal{O}(\log(n))$ with high probability

84b3f3, 135 lines

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());

struct Treap { // Implicit 0-idx
    struct Node {
        Node *l = NULL, *r = NULL;
        ll val, p;
        ll sz, sum, lazy;
        bool rev = false;
        Node(ll v) : val(v), p(rng()) {
            sz = 1, sum = val, lazy = 0;
        }
        void push() {
            if (lazy) {
                val += lazy, sum += lazy*sz;
                if (l) l->lazy += lazy;
                if (r) r->lazy += lazy;
            }
            if (rev) {
                swap(l, r);
                if (l) l->rev ^= 1;
                if (r) r->rev ^= 1;
            }
            lazy = 0, rev = 0;
        }
        void update() {
            sz = 1, sum = val;
            for(auto x : {l, r}) {
                if (x) {
                    x->push();
                    sz += x->sz;
                    sum += x->sum;
                }
            }
        }
    };

    Node* root;

    Treap() { root = NULL; }

    // copy constructor to remind the user to not copy the
        treap object
    Treap(const Treap& t) {
        throw logic_error("Nao copiar a Treap!");
    }

    ~Treap() { // deconstructor
        vector<Node*> q = {root};
        while (q.size()) {
            Node* x = q.back(); q.pop_back();
            if (!x) continue;
            q.pb(x->l), q.pb(x->r);
            delete x;
        }
    }

    ll size(Node* x) { return x ? x->sz : 0; }
    ll size() { return size(root); } // maybe useless line of
        code

    // Supposes that l < r when merging
    void merge(Node*& x, Node* l, Node* r) {
        if (!l or !r) return void(x = l ? l : r);
        l->push(), r->push();
        if (l->p > r->p) {
            merge(l->r, l->r, r);
            x = l;
        }
        else {
            merge(r->l, l, r->l);
            x = r;
        }
        x->update();
    }

    // split into [0, mid), [mid, n)
    // with size(left) = mid, size(right) = n-mid
    void split(Node* x, Node*& l, Node*& r, ll mid) {
        if (!x) return void(r = l = NULL);
        x->push();
        if (size(x->l) < mid) {
            split(x->r, x->r, r, mid - size(x->l) - 1);
            l = x;
        }
        else {
            split(x->l, l, x->l, mid);
            r = x;
        }
        x->update();
    }

    // insert new element with val=v into the rightmost
        position
    void insert(ll v) {
        Node* x = new Node(v);
        merge(root, root, x);
    }

    // get the query value for [l, r]
    ll query(ll l, ll r) {
        Node *L, *M, *R;
        split(root, M, R, r+1), split(M, L, M, l);
        ll ans = M->sum;
        merge(M, L, M), merge(root, M, R);
        return ans;
    }

    // increment value for [l, r] (not tested yet)
    void increment(ll l, ll r, ll s) {
        Node *L, *M, *R;
        split(root, M, R, r+1), split(M, L, M, l);
        M->lazy += s;
        merge(M, L, M), merge(root, M, R);
    }

    // reverses interval [l, r] to [r, l]
    void reverse(ll l, ll r) {
```

```
        Node *L, *M, *R;
        split(root, M, R, r+1), split(M, L, M, l);
        M->rev ^= 1;
        merge(M, L, M), merge(root, M, R);
    }

    // return in a vector all the elements in the treap, from
        left to right
    void inOrder(Node *u, vector<ll> &vec) {
        if (!u) return;
        u->push();
        // in-order
        inOrder(u->l, vec);
        vec.pb(u->val);
        inOrder(u->r, vec);
    }
    vector<ll> get() {
        vector<ll> vec;
        inOrder(root, vec);
        return vec;
    }
};
```

Dynamic Programming (3)

3.1 Longest Increasing Subsequence

If needed, the algorithm for LIS can be easily modified for the similar task of **Longest Non-Decreasing Subsequence**.

lis.cpp

Description: Computes the LIS size and also the auxiliar vector used to compute it. the LIS is STRICTLY INCREASING, but the given array can have duplicated values, the algorithm still works!

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

62b246, 29 lines

```
// returns {lis_size, vector<ll> mn[i] -> minimum last value of
    a LIS of size i}
pair<ll, vector<ll>> lis(vector<ll> x) {
    ll n = x.size();
    vector<ll> mn(n+1, INF); // mn[i] -> min value to achieve a
        LIS with size i
    mn[0] = -INF;
    for(auto val : x) {
        // find first element greater or equal than val
        ll pos = lower_bound(mn.begin(), mn.end(), val) - mn.
            begin();
        mn[pos] = val;
    }
    ll sz = lower_bound(mn.begin(), mn.end(), INF) - mn.begin()
        - 1;
    return {sz, mn};
}

// alternative code from USACO (not tested, but should work and
    vector index is off by an offset of -1)
int find_lis(vector<int> a) {
  vector<int> dp;
  for (int i : a) {
    int pos = lower_bound(dp.begin(), dp.end(), i) - dp.begin()
        ;
    if (pos == dp.size()) {
      // we can have a new, longer increasing subsequence!
      dp.push_back(i);
    } else {
      // oh ok, at least we can make the ending element smaller
      dp[pos] = i;
    }
  }
  return dp.size();
}
```

3.2 Divide-Conquer Optimization

Some dynamic programming problems have a recurrence of this form:

$$dp[i][k] = \min_{1 \le j \le i} dp[j-1][k-1] + C(j,i)$$

Where $dp[i][k]$ is the *min cost* considering the element up to i with exactly k partitions.

Additionally, $C(j,i)$ is the cost of the partition $[j,i]$ and $dp[i][k] = 0$ when $k = 0$.

Say $1 \le i \le n$ (1-idx) and $1 \le k \le m$, and evaluating C takes $O(1)$ time. Then the straightforward evaluation of the above recurrence is $O(mn^2)$. There are $n \times m$ states, and n transitions for each state.

Let $opt(i,k)$ be the value of j that minimizes the above expression, $opt(i,k)$ is the optimal splitting point.

Assuming that the cost function satisfies the *quadrangle inequality* ("wider is worse"), we can show that $opt(i-1,k) \le opt(i,k)$ for all i,k. This is known as the *monotonicity condition*. In other words, for a fixed k, the optimal splitting point $opt(i,k)$ increases as i increases.

This lets us solve for all states more efficiently. Say we computed $opt(i,k)$ for some fixed i and k. Then, for any $i' < i$, we know that $opt(i',k) \le opt(i,k)$. This means when computing $opt(i',k)$, we don't have to consider as many splitting points!

To minimize the runtime, we use this property and apply the idea behind *divide and conquer* and call the functions to solve recursively. Compute the $opt(l, mid)$ and using this value, solve $opt(l, mid-1)$ and $opt(mid+1, r)$.

By recursively keeping track of the lower and upper bounds on opt, we reach a $O(n \log n)$ runtime per k. Each possible value of $opt(i,k)$ only appears in $\log n$ different nodes.

Problems that can be solved:

- **Subarray Squares**: *cost* = square of the sum in each partition.
- **Houses and Schools**: each partition has a school at the *right endpoint*, and the cost is the accumulated walking time for each house, split in the middle, each half walking to the closest school. Precomputed $dp[i][1]$ considers the left border only walking right. Finally, *ans* is computed with $dp[i][m]$ + walking left of the remaining right border.

divide-conquer-dp.cpp

Description: Optimize an $O(mn^2)$ dp to $O(mnlog(n))$ using divide and conquer. cost function must have the quadrangle inequality ("wider is worse")

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

05e712, 122 lines

```
// Subarray Squares
int32_t main() { sws;
    // n elements, m partitions
    ll n, m; cin >> n >> m;

    vector<ll> vec(n+1);
    for(ll i=1; i<=n; i++) {
        cin >> vec[i];
    }

    vector<ll> ps(n+1, 0);
    for(ll i=1; i<=n; i++) {
        ps[i] = ps[i-1] + vec[i];
    }

    // cost the partition [l, r]
    auto cost = [&](ll l, ll r) {
        ll sum = ps[r] - ps[l-1];
        return sum * sum;
    };

    // dp[i][k] -> min cost; i = considered prefix, k = number
        of partitions
    // dp[i][k] = min{ dp[j-1][k-1] + cost([j, i]) }, j is the
        spliting point
    vector<vector<ll>> dp(n+1, vector<ll>(m+1, 0));

    // O(n log(n))
    function<void(ll, ll, ll, ll, ll)> solve = [&](ll k, ll l,
        ll r, ll optl, ll optr) {
        if (r < l) return;

        ll mid = (l+r)/2;
        dp[mid][k] = INF;
        ll opt = -1;

        for(ll j=optl; j<=min(mid, optr); j++) {
            ll val = dp[j-1][k-1] + cost(j, mid);
            if (val < dp[mid][k]) {
                dp[mid][k] = val;
                opt = j;
            }
        }

        solve(k, l, mid-1, optl, opt);
        solve(k, mid+1, r, opt, optr);
    };

    for(ll i=1; i<=n; i++) // one partition for all prefixes
        dp[i][1] = cost(1, i);

    for(ll k=2; k<=m; k++) { // compute the other [2, m]
        partitions
        solve(k, 1, n, 1, n);
```

```
    }

    cout << dp[n][m] << endl;
}

// Houses and Schools
int32_t main() { sws;
    ll n, m; cin >> n >> m;

    vector<ll> vec(n+1);
    for(ll i=1; i<=n; i++) {
        cin >> vec[i];
    }

    vector<ll> ps(n+1, 0), psl(n+1, 0), psr(n+1, 0);
    for(ll i=1; i<=n; i++) {
        ps[i]  = ps[i-1]  + vec[i];
        psl[i] = psl[i-1] + vec[i] * i;
        psr[i] = psr[i-1] + vec[i] * (n-i+1);
    }

    auto walk_left = [&](ll l, ll r) { // [0, 1, 2, ...]
        return psl[r] - psl[l-1] - l * (ps[r] - ps[l-1]);
    };

    auto walk_right = [&](ll l, ll r) { // [..., 2, 1, 0]
        return psr[r] - psr[l-1] - (n-r+1) * (ps[r] - ps[l-1]);
    };

    auto cost = [&](ll l, ll r) { // [0, 1, 2, ..., 2, 1, 0]
        ll mid = (l+r)/2;
        return walk_left(l, mid) + walk_right(mid+1, r);
    };

    // dp[i][k] -> min cost; i = considered prefix, k = number
        of partitions
    // dp[i][k] = min{ dp[j-1][k-1] + cost([j, i]) }, j is the
        spliting point
    vector<vector<ll>> dp(n+1, vector<ll>(m+1, 0));

    // O(n log(n))
    function<void(ll, ll, ll, ll, ll)> solve = [&](ll k, ll l,
         ll r, ll optl, ll optr) {
        if (r < l) return;

        ll mid = (l+r)/2;
        dp[mid][k] = INF;
        ll opt = -1;

        for(ll j=optl; j<=min(mid, optr); j++) {
            ll val = dp[j][k-1] + cost(j, mid); // j and mid
                 have schools
            if (val < dp[mid][k]) {
                dp[mid][k] = val;
                opt = j;
            }
        }

        solve(k, l, mid-1, optl, opt);
        solve(k, mid+1, r, opt, optr);
    };

    for(ll i=1; i<=n; i++) // one partition for all prefixes
        dp[i][1] = walk_right(1, i);

    for(ll k=2; k<=m; k++) { // compute the other [2, m]
         partitions
        solve(k, 1, n, 1, n);
    }

    ll ans = INF;
    for(ll i=1; i<=n; i++) {
        ans = min(ans, dp[i][m] + walk_left(i, n));
    }

    cout << ans << endl;
}
```

3.3 Knuth Optimization

Knuth's optimization, also known as the Knuth-Yao Speedup, is a special case of dynamic programming on ranges, that can optimize the time complexity of solutions by a linear factor, from $O(n^3)$ for standard range DP to $O(n^2)$.

3.3.1 Conditions

The Speedup is applied for transitions of the form:

$$dp(i,j) = \min_{i \le k < j} [dp(i,k) + dp(k+1,j) + C(i,j)].$$

Similar to divide and conquer DP, let $opt(i,j)$ be the value of k that minimizes the expression in the transition (opt is referred to as the "optimal splitting point" further in this article). The optimization requires that the following holds:

$$opt(i,j-1) \le opt(i,j) \le opt(i+1,j).$$

We can show that it is true when the cost function C satisfies the following conditions for $a \le b \le c \le d$:

$C(b,c) \le C(a,d)$;

$C(a,c) + C(b,d) \le C(a,d) + C(b,c)$ (the quadrangle inequality [QI]).

A common cost function that satisfies the above condition is the **sum of the values in a subaray**.

knuth.cpp

Description: Optimize $O(n^3)$ to $O(n^2)$ dp with transitions of finding a optimal division point k for [l, r].
Time: $\mathcal{O}(n^2)$ 8863a7, 37 lines

```
// dp[l][r] (inclusive) -> min cost
// opt[l][r] (inclusive) -> optimal spliting point k in l<=k<=r
ll dp[MAX][MAX], opt[MAX][MAX];

ll knuth(vector<ll> &vec) {
    // vec indexed with 1-idx, vec[0] = 0
    ll n = vec.size() - 1;

    vector<ll> ps(n+1, 0);
    for(ll i=1; i<=n; i++) {
        ps[i] = ps[i-1] + vec[i];
    }
    auto C = [&](ll l, ll r) {
        return ps[r] - ps[l-1];
    };

    for(ll i=1; i<=n; i++) {
        opt[i][i] = i;
    }

    for(ll l=n-1; l>=1; l--) {
        for(ll r=l+1; r<=n; r++) {
            ll mn = INF;
            ll cost = C(l, r);
            for(ll k=opt[l][r-1]; k<=min(r-1, opt[l+1][r]); k
                 ++) {
                ll aux = dp[l][k] + dp[k+1][r] + cost;
                if (aux <= mn) {
                    mn = aux;
                    opt[l][r] = k;
                }
            }
            dp[l][r] = mn;
        }
    }

    return dp[1][n];
}
```

3.4 Slope Optimizations

3.4.1 Convex Hull Trick

If multiple transitions of the DP can be seen as first degree polynomials (lines). CHT can be used to optimized it

Some valid functions:

$ax + b$

$cx^2 + ax + b$ (ignore cx^2 if c is independent)

cht-dynamic.cpp

Description: Dynamic version of CHT, thefore, one can insert lines in any order. There is no line removal operator
Time: $\mathcal{O}(\log N)$ per query and per insertion 707da4, 51 lines

```
// Convex Hull Trick Dinamico
//
// Para float, use LLINF = 1/.0, div(a, b) = a/b
//
// update(x) atualiza o ponto de intersecao da reta x
// overlap(x) verifica se a reta x sobrepoe a proxima
// add(a, b) adiciona reta da forma ax + b
// query(x) computa maximo de ax + b para entre as retas
// se quiser computar o minimo, eh soh fazer (-a)x + (-b)
//
// O(log(n)) amortizado por insercao
// O(log(n)) por query

struct Line {
  mutable ll a, b, p;
  bool operator<(const Line& o) const { return a < o.a; }
  bool operator<(ll x) const { return p < x; }
};

struct DynamicCHT : multiset<Line, less<>> {
  ll div(ll a, ll b) {
    return a / b - ((a ^ b) < 0 and a % b);
  }

  void update(iterator x) {
    if (next(x) == end()) x->p = LLINF;
    else if (x->a == next(x)->a) x->p = x->b >= next(x)->b ?
         LLINF : -LLINF;
    else x->p = div(next(x)->b - x->b, x->a - next(x)->a);
  }
```

```
  bool overlap(iterator x) {
    update(x);
    if (next(x) == end()) return 0;
    if (x->a == next(x)->a) return x->b >= next(x)->b;
    return x->p >= next(x)->p;
  }

  void add(ll a, ll b) {
    auto x = insert({a, b, 0});
    while (overlap(x)) erase(next(x)), update(x);
    if (x != begin() and !overlap(prev(x))) x = prev(x), update
        (x);
    while (x != begin() and overlap(prev(x)))
      x = prev(x), erase(next(x)), update(x);
  }

  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.a * x + l.b;
  }
};
```

3.4.2 Li-chao Tree

Works for any type of function that has the **transcending property**:

Given two functions $f(x)$, $g(x)$ of that type, if $f(t)$ is greater than/smaller than $g(t)$ for some $x = t$, then $f(x)$ will be greater than/smaller than $g(x)$ for $x > t$. In other words, once $f(x)$ "win/lose" $g(x)$, $f(x)$ will continue to "win/lose" $g(x)$.

The most common one is the line function: $ax + b$

Due to the segment tree structure, Li-Chao tree also supports adding line **segments**.

lichao.cpp

Description: Basic Li-chao tree implementation, in which every node in a segment tree contains the one needed line that has the min/max query
Time: $\mathcal{O}(\log N)$ for add-line(), query(); $\mathcal{O}(\log^2 N)$ for add-segment; N = max-value considered of X

c41630, 69 lines

```
// Lichao tree for minimum query
// to adapt to max query:
// +INF -> -INF
// min() -> max()
// '>' -> '<' in add_line()
struct Lichao {
    struct Line {
        ll a = 0, b = +INF; // ax + b
        ll operator()(ll x) {
            return a*x + b;
        }
    };

    ll L = 0, R;
    vector<ll> v;
    vector<Line> tree;

    Lichao(ll n) : R(n), v(n+1), tree(4*(n+1)) {}

    void add_line(Line line, ll l, ll r, ll i) {
        ll mid = (l+r)/2;
        if (tree[i](mid) > line(mid)) {
            swap(tree[i], line);
        }
        if (tree[i](l) > line(l)) {
            add_line(line, l, mid, 2*i);
        }
        if (tree[i](r) > line(r)) {
            add_line(line, mid+1, r, 2*i+1);
        }
    }
    void add_line(Line line) {
        add_line(line, L, R, 1);
    }

    void add_segment(Line line, ll left, ll right, ll l, ll r,
         ll i) {
        if (left <= l and r <= right) {
            add_line(line, l, r, i);
            return;
        }
        ll mid = (l+r)/2;
        if (left <= mid) {
            add_segment(line, left, right, l, mid, 2*i);
        }
        if (mid < right) {
            add_segment(line, left, right, mid+1, r, 2*i+1);
        }
    }
    void add_segment(Line line, ll left, ll right) {
        add_segment(line, left, right, L, R, 1);
    }

    ll query(ll x, ll l, ll r, ll i) {
        if (l == r) return tree[i](x);
        ll mid = (l+r)/2;
        ll ans = tree[i](x);

        if (x <= mid) {
            ans = min(ans, query(x, l, mid, 2*i));
        }
        else {
            ans = min(ans, query(x, mid+1, r, 2*i+1));
        }
        return ans;
    }
    ll query(ll x) {
        return query(x, L, R, 1);
    }
};
```

s

3.4.3 Slope Trick

You are given an array of n integers. You want to modify the array so that it is non-decreasing, i.e., every element is at least as large as the previous element. On each move, you can increase or decrease the value of any element by one. What is the minimum number of moves required?

Observation: It is also possible to solve the problem of modifying the array to stricly increasing.

slope-trick.cpp

Description: Using Slope trick, compute the min cost to modify arry to be non-decreasing
Time: $\mathcal{O}(n \log n)$

25b6fc, 58 lines

```
// funcao f_i(x) = custo de deixar todo mundo ate i
// nao decrescente e <= x

// os pontos em changepoints sao os pontos da
// piecewise linear function convexa

// eu calculo g_i(x) = custo de deixar todo mundo ate i
// nao decrescente e v[i] == x

// entao f_i(x) = min(g_i(t) pra t <= x)

// podemos escrever gi(x) = fi-1(x) + |x-v[i]|
// entao a gente ta somando as funcoes e gerando outra convexa

// a resposta vai armazenar o custo (coord y) do opt
// e o topo do change_points vai ser o opt atual

// se opt < v[i] entao a gente calcula o g_i e o novo opt
// vai ser v[i]

// se opt > v[i] entao o slope entre opt e anterior opt vai
     ficar reto
// (este anterior opt podendo ser o v[i] que vai ser inserido),
// entao basta retirar o ultimo opt e teremos de novo a
     resposta
// neste caso devemos aumentar o custo do opt, que vai ser por
// (opt - v[i]) (so olhar a funcao em V do || e a convexa do fi
     -1)

// o v[i] vai ser inserido varias vezes no change_points
// pra denotar a inclinacao no slope dele

int32_t main() { sws;

    ll n; cin >> n;

    vector<ll> v(n);
    for(ll i = 0; i < n; i++) {
        cin >> v[i];
        // to change the problem
        // from increasing to non-decreasing
        // v[i] -= i;
    }

    priority_queue<ll> change_points;
    change_points.push(-INF);
    ll ans = 0;

    for(ll i = 0; i < n; i++) {
        ll opt = change_points.top();
        change_points.push(v[i]);

        if(opt > v[i]) {
            ans += opt - v[i];
            change_points.push(v[i]);
            change_points.pop();
        }
    }

    cout << ans << endl;
}
```

3.5 SOS DP

Sum over Subsets DP (SOS DP) computes how many elements there are for each mask which are a subset of this mask.

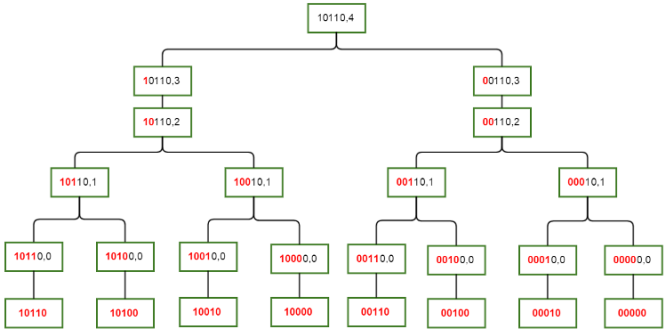
This can be modified for other operations in which the subset contributes for the mask . *Example:*

1001 if a subset of **1101**;

0001 if a subset of **1101**;

1100 if a subset of **1101**;

1101 if a subset of **1101**;



sos-dp.cpp
Description: Efficiently compute a bitmask dp, in which a subset of this bitmask contributes for the value of this bitmask.
Time: $\mathcal{O}\left(2^N N\right)$, N = number of bits
19e50a, 35 lines

```
// problem: Given a list of n integers, your task is to
     calculate for each element x:
// the number of elements y such that x | y = x
// the number of elements y such that x & y = x
// the number of elements y such that x & y != 0

const ll LOGMAX = 20;
ll dp[1 << LOGMAX];
ll dp2[1 << LOGMAX];

int32_t main(){ sws;
    ll n; cin >> n;

    vector<ll> a(n);
    for(auto &val : a) cin >> val;

    ll full = (1LL << LOGMAX) - 1;

    for(auto val : a) dp2[full^val] += 1;
    for(auto val : a) dp[val] += 1;

    for(ll b=0; b<LOGMAX; b++) {
        for(ll mask=0; mask<(1LL<<LOGMAX); mask++) {
            if (mask & (1LL << b)) {
                dp[mask] += dp[mask ^ (1LL << b)];
                dp2[mask] += dp2[mask ^ (1LL << b)];
            }
        }
    }

    for(auto val : a) {
        cout << dp[val] << " ";
        cout << dp2[full ^ val] << " ";
        cout << n - dp[full^val] << endl;
    }
}
```

3.6 Bit optimization

use popcnt pragma!!

```
#pragma GCC target("popcnt")
```

3.6.1 Operations

intersection	$a \cap b$	a&b
union	$a \cup b$	a|b
complement	$\overline{a}$	a
difference	$a - b$	a&(b)

- **__builtin_clz(x)**: the number of zeros at the beginning of the number
- **__builtin_ctz(x)**: the number of zeros at the end of the number
- **__builtin_popcount(x)**: the number of ones in the number
- **__builtin_parity(x)**: the parity (even or odd) of the number of ones
- **LSB(i)**: ((i) & -(i))
- **MSB(i)**: (63 - __builtin_clzll(i)), for ll

3.6.2 Bitset

Bitset are very convenient for bitwise operations. Beside common operators, there are other useful ones already built in:

- **bitset <k> bs(str)**: create a bitset of size k from a binary string representation
- **bitset <k> bs(num)**: create a bitset of size k from a integer representation
- **str = bs.to_string()**: return the binary string representation of the bitset
- **num = bs.to_ullong()()**: return the unsigned integer representation of the bitset
- **bs._Find_first()**: returns the first set bit (from LSB to MSB)
- **bs._Find_next(idx)**: returns the next set bit after idx (not including idx of course)

Note that, if there isn't any set bit after idx, BS._Find_next(idx) will return BS.size(); same as calling BS._Find_first() when bitset is clear; One can use **bs._Find_next(idx-1)** to include idx. The function does accept negative index.

The complexity of bitwise operations for the bitset is $O(\frac{size}{32})$ or $O(\frac{size}{64})$, depending on the architecture of the computer.

3.6.3 Problems

- **Hamming Distance:** When comparing two binary strings of size k, if the size of the strings are small enough, just represent them as integers (uint or ulong) and do $\_\_builtin\_popcount(a\ \hat{}\ b)$ to compute the hamming distance in $O(1)$ instead of $O(k)$.
- **Counting subgrids:** If the desired size if not small enough, divide into continuous segments of acceptable sizes (such as k=64 for unsigned long long). Then, the complexity of $O(N)$ can be reduced to $O(N/64)$. For more versatility, and huge sizes, one can use bitset¡k¿ directly, but it is a little bit slower.

Game theory (4)

4.1 Classic Game

- There are n piles (heaps), each one with x_i stones.
- Each turn, a players must remove t stones (non-zero) from a pile, turning x_i into y_i.
- The game ends when it's impossible to make any more moves and the player without moves left lose.

4.2 Bouton's Theorem

Let s be the xor-sum value of all the piles sizes, a state $s = 0$ is a losing position and a state $s! = 0$ is a winnig position

4.2.1 Proof

All wining positions will have at least one valid move to turn the game into a losing position.

All losing positions will only have moves that turns the game into winning positions (except the base case when there are no piles left and the player already lost)

4.3 DAG Representation

Consider all game positions or states of the game as **Vertices** of a graph

Valid moves are the transition between states, therefore, the directed **Edges** of the graph

If a state has no outgoing edges, it's a dead end and a losing state (degenerated state).

If a state has only edges to winning states, therefore it is a losing state.

if a state has at least one edge that is a losing state, it is a winning state.

4.4 Sprague-Grundy Theorem

Let's consider a state u of a two-player impartial game and let v_i be the states reachable from it.

To this state, we can assign a fully equivalent game of Nim with one pile of size x. The number x is called the **Grundy value or nim-value or nimber** of the state u.

If **all transitions** lead to a *winning state*, the current state must be a *losing state* with nimber 0.

If **at least one transition** lead to a *losing state*, the current state must be a *winning state* with nimber ¿ 0.

The **MEX** operator satisfies both condition above and can be used to calculate the nim-value of a state:

$nimber_u$ = MEX of all $nimber_{v_i}$

Viewing the game as a DAG, we can gradually calculate the Grundy values starting from vertices without outgoing edges (nimber=0).

Note that the MEX operator **garantees** that all nim-values smaller than the considered nimber can be reached, which is essentialy the nim game with a single heap with pile size = nimber.

There are only two operations that are used when considering a Sprague-Grundy game:

4.4.1 Composition

XOR operator to compose sub-games into a single composite game

When a game is played with multiple sub-games (as nim is played with multiple piles), you are actually choosing one sub-game and making a valid move there (choosing a pile and subtracting a value from it).

The final result/winner will depend on all the sub-games played. Because you need to play all games.

To compute the final result, one can simply consider the XOR of the nimbers of all sub-games.

4.4.2 Decomposition

MEX operator to compute the nimber of a state that has multiple transitions to other states

A state with nimber x can be transitioned (decomposed) into all states with nimber $y < x$

Nevertheless a state may reach several states, only a single one will be used during the game. This shows the difference between **states** and **sub-games**: All sub-games must be played by the players, but the states of a sub-game may be ignored.

To compute the mex of a set efficiently:

mex.cpp

Description: Compute MEX efficiently by keeping track of the frequency of all existent elements and also the missing ones
Time: $\mathcal{O}(\log N)$ per addition/removal, $\mathcal{O}(1)$ to get mex value, $\mathcal{O}(N \log(N))$ to initialize

d6f2b9, 27 lines

```
struct MEX {
    map<ll, ll> freq;
    set<ll> missing;

    // initialize set with values up to {max_valid_value}
        inclusive
    MEX(ll max_valid_value) { // O(n log(n))
        for(ll i=0; i<=max_valid_value; i++)
            missing.insert(i);
    }

    ll get() { // O(1)
        if (missing.empty()) return 0;
        return *missing.begin();
    }

    void remove(ll val) { // O(log(n))
        freq[val]--;
        if (freq[val] == 0)
            missing.insert(val);
    }

    void add(ll val) { // O(log(n))
        freq[val]++;
        if (missing.count(val))
            missing.erase(val);
    }
};
```

4.5 Variations and Extensions

4.5.1 Nim with Increases

Consider a modification of the classical nim game: a player can now add stones to a chosen pile instead of removing.

Note that this extra rule needs to have a restriction to keep the game acyclic (finite game).

Lemma: This move is not used in a winnig strategy and can be ignored.

Proof: If a player adds t stones in a pile, the next player just needs to remove t stones from this pile.

Considering that the game is finite and this ends sooner or later.

Example: If the set of possible outcomes for a state is 0, 1, 2, 7, 8, 9. The nimber is 3, because the MEX is 3, which is the smallest nim-value you can't transition into and also you can transition to all smaller nim-values.

Note that 7, 8, 9 transitions can be ignored, because you can simply revert the play by subtracting the same amount.

4.6 Misère Game

In this version, the player who takes the last object loses. To consider this version, simply swap the winning and losing player of the normal version.

4.7 Staircase Nim

4.7.1 Description

In Staircase Nim, there is a staircase with n steps, indexed from 0 to n-1. In each step, there are zero or more coins. Two players play in turns. In his/her move, a player can choose a step ($i > 0$) and move one or more coins to step below it (i-1). The player who is unable to make a move lose the game. That means the game ends when all the coins are in step 0.

4.7.2 Strategy

We can divide the steps into two types, odd steps, and even steps.

Now let's think what will happen if a player A move x coins from an even step(non-zero) to an odd step. Player B can always move these same x coins to another even position and **the state of odd positions aren't affected**

But if player A moves a coin from an odd step to an even step, similar logic won't work. Due to the degenerated case, there is a situation when x coins are moved from stair 1 to 0, and player B can't move these coins from stair 0 to -1 (not a valid move).

From this argument, we can agree that coins in even steps are useless, they don't interfere to decide if a game state is winning or losing.

Therefore, the staircase nim can be visualized as a simple nim game with only the odd steps.

When stones are sent from an odd step to an even step, it is the same as removing stones from a pile in a classic nim game.

And when stones are sent from even steps to odd ones, it is the same as the increasing variation described before.

4.8 Grundy's Game

Initially there is only one pile with x stones. Each turn, a player must divide a pile into two non-zero piles with different sizes. The player who can't do any more moves loses.

s

4.8.1 Degenerate (Base) States

x = 1 (nim-val = 0) (losing)

x = 2 (nim-val = 0) (losing)

4.8.2 Other States

nim-val = MEX (all transitions)

Examples

x = 3:

```
{2, 1} -> (0) xor (0) -> 0

nim-val = MEX({0}) = 1
```

x = 4:

```
{3, 1} -> (1) xor (0) -> 1

nim-val = MEX({1}) = 0
```

x = 5:

```
{4, 1} -> (0) xor (0) -> 0
{3, 2} -> (1) xor (0) -> 1

nim-val = MEX({0, 1}) = 2
```

x = 6:

```
{5, 1} -> (2) xor (0) -> 2
{4, 2} -> (0) xor (0) -> 0

nim-val = MEX({0, 2}) = 1
```

Important observation: All nimbers for (n $\geq$ 2000) are non-zero. (missing proof here and testing for values above 1e6).

4.9 Insta-Winning States

Classic nim game: if **all** piles become 0, you lose. (no more moves)

Modified nim game: if **any** pile becomes 0, you lose.

To adapt to this version of nim game, we create insta-winning states, which represents states that have a transition to any empty pile (will instantly win). Insta-winning states must have an specific nimber so they don't conflict with other nimbers when computing. A possible solution is nimber=INF, because no other nimber will be high enough to cause conflict.

Because of this adaptation, we can now ignore states with empty piles, and consider them with ($nullvalue = -1$). And the ($nimber = 0$) now represents the states that only have transitions to insta-winning states.

After this, beside winning states and losing states, we have added two new categories of states (insta-winning and empty-pile). Notice that:

```
empty-pile <- insta-winning <- nimber(0)
```

Therefore, we have returned to the classical nim game and can proceed normally.

OBS: *Empty piles* (wasn't empty before) ($nimber = -1$) is different from *Non-existent piles* (never existed) (nimber = 0)

Usage Example:
https://codeforces.com/gym/101908/problem/B

4.10 References

https://cp-algorithms.com/game_theory/sprague-grundy-nim.html

https://codeforces.com/blog/entry/66040

https://brilliant.org/wiki/nim/

Geometry (5)

5.1 Point Struct

point.cpp
Description: Point struct for point operations, supports floating points and integers
Time: $\mathcal{O}(1)$
7e11ab, 43 lines

```
const ld EPS = 1e-9;

// T can be int, long long, float, double, long double
template<class T> bool eq(T a, T b) {
    if (is_integral<T>::value) return a == b;
    else return abs(a-b) <= EPS;
}

template<class T> struct P {
    T x, y;
    ll id; // (optional)

    P(T xx=0, T yy=0): x(xx), y(yy) {}

    P operator +(P const& o) const { return { x+o.x, y+o.y }; }
    P operator -(P const& o) const { return { x-o.x, y-o.y }; }
    P operator *(T const& t) const { return { x*t, y*t }; }
    P operator /(T const& t) const { return { x/t, y/t }; }
    T operator *(P const& o) const { return x*o.x + y*o.y; }
    T operator ^(P const& o) const { return x*o.y - y*o.x; }

    bool operator <(P const& o) const { // enables sorting, set, etc
        return (eq(x, o.x) ? y < o.y : x < o.x);
    }

    bool operator ==(P const& o) const {
        return eq(x, o.x) and eq(y, o.y);
    }

    bool operator !=(P const& o) const {
        return !(*this == o);
    }

    friend istream& operator >>(istream& in, P &p) {
        return in >> p.x >> p.y;
    }

    friend ostream& operator <<(ostream& out, P const& p) {
        return out << p.x << ' ' << p.y;
    }
};
using point = P<ll>;
// using point = P<ld>;
```

5.2 Line Struct

line.cpp
Description: Line struct for line operations
Time: $\mathcal{O}(1)$
5f33bd, 29 lines

```
template<class T> struct L {
    point p1, p2;
    T a, b, c; // ax+by+c = 0;

    // y-y1 = ((y2-y1)/(x2-x1))(x-x1)
    L(point pp1=0, point pp2=0) : p1(pp1), p2(pp2) {
        a = p1.y - p2.y;
        b = p2.x - p1.x;
        c = p1 ^ p2;
    }

    T eval(point p) {
        return a*p.x + b*p.y + c;
    }

    bool inside(point p) { // reta
        return eq(eval(p), T(0));
    }

    point normal() {
        return point(a, b);
    }

    bool insideSeg(point p) { // segmento [p1, p2]
        return ( ((p1-p) ^ (p2-p)) == 0 and ((p1-p) * (p2-p))
            <= 0 );
    }
};
using line = L<ll>;
// using line = L<ld>;
```

5.3 Manhattan Minimum Spanning Tree

Also called the rectilinear or L1 Minimum Spanning Tree problem.

manhattanMST.cpp
Description: Given N points, returns up to 4*N edges, which are guaranteed to contain a minimum spanning tree for the graph with edge weights w(p, q) = —p.x - q.x— + —p.y - q.y—. Edges are in the form (distance, src, dst). Use a standard MST algorithm on the result to find the final MST.
Time: $\mathcal{O}(N \log N)$
00e093, 25 lines

```
// requires point struct, at least the constructor and operator -

vector<array<ll, 3>> manhattanMST(vector<point> ps) {
    vector<ll> id(size(ps));
    iota(id.begin(), id.end(), 0);
    vector<array<ll, 3>> edges;
    for(ll k=0; k<4; k++) {
        sort(id.begin(), id.end(), [&](ll i, ll j) {
            return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;
        });

        map<ll, ll> sweep;
        for (ll i : id) {
            for (auto it = sweep.lower_bound(-ps[i].y); it !=
                 sweep.end(); sweep.erase(it++)) {
                ll j = it->ss;
                point d = ps[i] - ps[j];
                if (d.y > d.x) break;
```

```
                edges.pb({d.y + d.x, i, j});
            }
            sweep[-ps[i].y] = i;
        }
        for (point& p : ps) if (k & 1) p.x = -p.x; else swap(p.
            x, p.y);
    }
    return edges;
}
```

Graph (6)

6.1 Fundamentals

Curious Property of DFS:

- Given an undirected graph, assign each node to a set A.
- Run a Depth-First-Search starting at any node.
- Whenever the DFS visits a new node N, remove N from A and add it to the path set P.
- Whenever the DFS backtracks from node N, remove N from the path and add it to set B.
- Repeat until $|A| = |B|$. Which **will always occur**, because, in each operation, A decreases by one and B keeps it value. Or, B decreases by one and A keeps it value.
- The DFS guarantees that A and B never have neighbouring nodes. Because the set of nodes in Path P separates them.

dfs.cpp
Description: Simple DFS template with lambda syntax
Time: $\mathcal{O}(V+E)$
af867f, 19 lines

```
int32_t main(){ sws;
    // compute cardinality of each subtree
    vector<vll> g(n);
    vector<ll> card(n);
    vector<bool> vis(n); // redundant here

    function<ll (ll, ll)> dfs = [&](ll u, ll p) -> ll {
        if (vis[u]) return;
        vis[u] = 1;

        card[u] += 1;
        for(auto v : g[u]) if (v != p) {
            card[u] += dfs(v, u);
        }
        return card[u];
    };

    dfs(1, -1);
}
```

bfs.cpp
Description: Simple BFS template
Time: $\mathcal{O}(V+E)$
7bed46, 34 lines

```
vector<vll> g(n);
vector<ll> d(n);
vector<bool> vis(n);

void bfs(ll src, ll sink) {
    queue<ll> q;
    q.push(src);
    d[src] = 0;
    vis[src] = 1;

    while(!q.empty()) {
        auto u = q.front(); q.pop();

        // add here a special break condition if needed, ex:
        if (u == sink) break;

        for(auto v : g[u]) {

            // each v is added to queue only once
            // due to checking visited inside for(auto v : g[u
                ])
            // and setting vis[v] = 1 before pushing to queue
            if (!vis[v]) {
                vis[v] = 1;
                d[v] = d[u] + 1;
                q.push(v);
            }

            else { // already added to queue, but there may be
                a shorter path
                d[v] = min(d[v], d[u] + 1);
            }
        }
    }
}
```

6.2 Network flow

In optimization theory, maximum flow problems involve finding a feasible flow through a flow network that obtains the maximum possible flow rate.

dinic.cpp
Description: Run several bfs to compute the residual graph until a max flow configuration is discovered
Time: General Case, $\mathcal{O}(V^2E)$; Unit Capacity, $\mathcal{O}\left((V+E)\sqrt{E}\right)$; Bipartite and unit capacity, $\mathcal{O}\left((V+E)\sqrt{V}\right)$
dea1b7, 86 lines

```
// remember to duplicate vertices for the bipartite graph
// N = number of nodes, including sink and source
const ll N = 700;

struct Dinic {
    struct Edge {
        ll from, to, flow, cap;
    };
    vector<Edge> edges;

    vector<ll> g[N];
    ll ne = 0, lvl[N], vis[N], pass;
    ll qu[N], px[N], qt;

    ll run(ll s, ll sink, ll minE) {
        if (s == sink) return minE;
        ll ans = 0;
        for(; px[s] < (int)g[s].size(); px[s]++){
            ll e = g[s][ px[s] ];
            auto &v = edges[e], &rev = edges[e^1];
            if( lvl[v.to] != lvl[s]+1 || v.flow >= v.cap)
                continue;
            ll tmp = run(v.to, sink, min(minE, v.cap - v.flow))
                ;
            v.flow += tmp, rev.flow -= tmp;
            ans += tmp, minE -= tmp;
            if (minE == 0) break;
        }
        return ans;
    }

    bool bfs(ll source, ll sink) {
        qt = 0;
        qu[qt++] = source;
        lvl[source] = 1;
        vis[source] = ++pass;
        for(ll i=0; i<qt; i++) {
            ll u = qu[i];
            px[u] = 0;
            if (u == sink) return 1;
            for(auto& ed :g[u]) {
                auto v = edges[ed];
                if (v.flow >= v.cap || vis[v.to] == pass)
                    continue;
                vis[v.to] = pass;
                lvl[v.to] = lvl[u]+1;
                qu[qt++] = v.to;
            }
        }
        return false;
    }

    ll flow(ll source, ll sink) { // max_flow
        reset_flow();
        ll ans = 0;
        while(bfs(source, sink))
            ans += run(source, sink, LLINF);
        return ans;
    }

    void addEdge(ll u, ll v, ll c, ll rc = 0) { // c = capacity
        , rc = retro-capacity;
        Edge e = {u, v, 0, c};
        edges.pb(e);
        g[u].pb(ne++);
        e = {v, u, 0, rc};
        edges.pb(e);
        g[v].pb(ne++);
    }

    void reset_flow() {
        for (ll i=0; i<ne; i++) edges[i].flow = 0;
        memset(lvl, 0, sizeof(lvl));
        memset(vis, 0, sizeof(vis));
        memset(qu, 0, sizeof(qu));
        memset(px, 0, sizeof(px));
        qt = 0; pass = 0;
    }

    // cut set cost = minimum cost = max flow
    // cut set is the set of edges that, if removed,
    // will disrupt flow from source to sink and make it 0.
    vector<pll> cut() {
        vector<pll> cuts;
        for (auto [from, to, flow, cap]: edges)
            if (flow == cap and vis[from] == pass and vis[to] <
                pass and cap > 0)
                cuts.pb({from, to});
        return cuts;
    }
};
```

dinitz.cpp

Description: This second version may be slower due to dynamic allocation, queue, etc but it's more readable, more memory efficient

Time: General Case, $\mathcal{O}\left(V^2E\right)$; Unit Capacity, $\mathcal{O}\left((V+E)\sqrt{E}\right)$; Bipartite and unit capacity, $\mathcal{O}\left((V+E)\sqrt{V}\right)$

ef9863, 90 lines

```
struct Dinitz {
    struct Edge { // u -> v
        ll u, v, cap, flow=0; // u is redundant, but nice for
             some problems
    };

    ll n, src, sink;
    vector<Edge> edges;
    vector<vector<ll>> g;
    vector<ll> level, ptr;

    Dinitz(ll nn) : n(nn+10), g(n) {
        src = n-2;
        sink = n-1;
    }
    Dinitz(ll nn, ll s, ll t) : n(nn+10), g(n) {
        src = s;
        sink = t;
    }

    void addEdge(ll u, ll v, ll cap, ll rcap = 0) { // rcap =
         retrocapacity for bidiretional edges
        g[u].push_back( (ll)edges.size() );
        edges.push_back({u, v, cap});
        g[v].push_back( (ll)edges.size() );
        edges.push_back({v, u, rcap});
    }

    bool bfs() {
        level.assign(n, -1); // not vis
        level[src] = 0;
        queue<ll> q;
        q.push(src);
        while (!q.empty()) {
            ll u = q.front(); q.pop();
            for (auto eid : g[u]) {
                auto e = edges[eid];
                if (e.flow >= e.cap or level[e.v] != -1)
                    continue;
                level[e.v] = level[u] + 1;
                q.push(e.v);
            }
        }
        return level[sink] != -1;
    }

    ll dfs(ll u, ll f) {
        if (f == 0 or u == sink) return f;
        for (ll &i = ptr[u]; i < (ll)g[u].size(); i++) {
            ll eid = g[u][i];
            auto &e = edges[eid];
            if(e.flow >= e.cap or level[u]+1 != level[e.v])
                continue;
            ll newf = dfs(e.v, min(f, e.cap - e.flow));
            if (newf == 0) continue;
            e.flow += newf;
            edges[eid^1].flow -= newf;
            return newf;
        }
        return 0;
    }

    ll max_flow = 0;
    ll flow(bool reset_flow = true) {
        if (reset_flow) {
            max_flow = 0;
            for(ll u=0; u<n; u++) {
                for(auto eid : g[u]) {
                    auto &e = edges[eid];
                    e.flow = 0;
                }
            }
        }
        while (bfs()) {
            ptr.assign(n, 0);
            while (ll newf = dfs(src, INF))
                max_flow += newf;
        }
        return max_flow;
    }

    // minimum cut set cost = minimum cost = max flow
    // minimum cut set is the minimum set of edges that, if
         removed,
    // will disrupt flow from source to sink and make it 0.
    vector<pll> cut() {
        vector<pll> cuts;
        for (auto [u, v, cap, flow]: edges) {
            if (level[u] != -1 and level[v] == -1) {
                cuts.pb({u, v});
            }
        }
        return cuts;
    }
};
```

6.2.1 Matching with Flow

By modeling a bipartite graph, with some Vertices (that will choose a match) to be on the L graph and some Vertices (that will be chosen) on the R. Set the correct capacities for these edges and also for the edges that connects the sink and source. After this modeling and running the dinic max flow algorithm, one will generate a possible matching (if it is possible).

A special case of matching is the perfect matching, which includes all vertices from the bipartite graph L and R.

A maximum matching has the maximum cadinality. A perfect matching is a maximum matching. But the opposite is not necessarity true.

It's possible to access dinic.edges, which is a vector that contains all edges and also its respective attributes, like the *flow* passing through each edge. Remember to consider that negative flow exist for reverse edges.

6.2.2 Minimum Cut

In computer science and optimization theory, the max-flow min-cut theorem states that, in a flow network, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in a minimum cut, i.e., the smallest total weight of the edges which if removed would disconnect the source from the sink.

Let's define an s-t cut C = (S-component, T-component) as a partition of V ∈ G such that source s ∈ S-component and sink t ∈ T-component. Let's also define a cut-set of C to be the set (u, v) ∈ E — u ∈ S-component, v ∈ T-component such that if all edges in the cut-set of C are removed, the Max Flow from s to t is 0 (i.e., s and t are disconnected). The cost of an s-t cut C is defined by the sum of the capacities of the edges in the cut-set of C.

The by-product of computing Max Flow is Min Cut! After Max Flow algorithm stops, we run graph traversal (DFS/BFS) from source s again. All reachable vertices from source s using positive weighted edges in the residual graph belong to the S-component. All other unreachable vertices belong to the T-component. All edges connecting the S-component to the T-component belong to the cut-set of C. The Min Cut value is equal to the Max Flow value. This is the minimum over all possible s-t cuts values.

6.2.3 Minimum Vertex Cover

The **Konig's Theorem** describes an equivalence between the maximum matching problem and the minimum vertex cover problem in bipartite graphs.

Therefore, the value for the maximum flow in a bipartite graph is the same value as the number of nodes in a minimum vertex cover.

To retrieve the set of vertices of the minimum vertex cover:

- Give orientation to the edges, matched edges start from the right side of the graph to the left side, and free edges start from the left side of the graph to the right side.
- Run DFS from unmatched nodes of the left side, in this traversal some nodes will become visited, others will stay unvisited.
- The MVC nodes are the visited nodes from the right side, and unvisited nodes from the left side.

$$MVC = Visited_{Right} \cup Unvisited_{Left}$$

min-vertex-cover.cpp

Description: computes the min vertex cover for a bipartite graph matched with dinitz

Time: $\mathcal{O}(Elog(E))$ 963b5e, 55 lines

```
// a vertex cover is a set of vertices that contains
// at least one endpoint for each edge in the bipartite match
// A vertex cover in minimum if no other vertex cover has fewer
      vertices.
// only for bipartite graphs
vector<ll> minVertexCover(Dinitz &dinitz) {
    ll n = dinitz.n;

    vector<vector<ll>> g(n);
    set<ll> left, right; // unique
    vector<bool> matched(n);

    for(auto e : dinitz.edges) {
        if (e.u == dinitz.src or e.u == dinitz.sink) continue;
        if (e.v == dinitz.src or e.v == dinitz.sink) continue;
        if (e.cap > 0) { // not retro edge

            left.insert(e.u);
            right.insert(e.v);

            if (e.flow == e.cap) {
                // orient matched edges from right to left
                g[e.v].pb(e.u);
                matched[e.u] = 1;
                matched[e.v] = 1;
            }
            else {
                // orient non-matched edges from left to right
                g[e.u].pb(e.v);
            }
        }
    };

    vector<bool> vis(n, 0);
    function<void (ll)> dfs = [&](ll u) {
        vis[u] = 1;
        for(auto v : g[u])
            if (!vis[v])
                dfs(v);
    };

    for(auto l : left) if (!matched[l]) {
        dfs(l);
    }

    vector<ll> ans;
    for(auto l : left) if (!vis[l]) {
        ans.pb(l);
    }
    for(auto r : right) if (vis[r]) {
        ans.pb(r);
    }

    // remember, right nodes ids are dislocated by an offset
    return ans;
}
```

6.2.4 Maximum Independent Set

A **Independent Set** is a subset of nodes, in which all pairs u, v in the subset are not adjacent (There is no direct edge between nodes u and v).

A **Maximum Independent Set** is a *Independent Set* with maximum cardinality;

The **Maximum Independent Set** is complementar to the **Minimum Vertex Cover**.

$$MaxIS = all_{Vertices} \setminus MVC$$

Therefore, to acquire the **Maximum Independent Set**, run the MVC algorithm and subtract them from the set of vertices and it will end up with the maxIS.

6.3 Minimum Cost Matching

6.3.1 Minimum Cost with Dinitz

min-cost-dinitz.cpp

Description: change bfs to spfa to attribute a weight for the edges

Time: SPFA is $\mathcal{O}(E)$ at average and $\mathcal{O}(VE)$ in the worst case a18e95, 90 lines

```
struct Dinitz {
    struct Edge { // u -> v
        ll u, v, cost, cap, flow=0;
    };

    ll n, src, sink;
    vector<Edge> edges;
    vector<vector<ll>> g;
    vector<ll> dist, ptr; // uses dist instead of level

    Dinitz(ll nn) : n(nn+10), g(n) {
        src = n-2;
        sink = n-1;
    }
    Dinitz(ll nn, ll s, ll t) : n(nn+10), g(n) {
        src = s;
        sink = t;
    }

    void addEdge(ll u, ll v, ll cost, ll cap, ll rcap = 0) { //
          rcap = retrocapacity for bidiretional edges
        g[u].push_back( (ll)edges.size() );
        edges.push_back({u, v, cost, cap});
        g[v].push_back( (ll)edges.size() );
        edges.push_back({v, u, -cost, rcap});
    }

    bool spfa() {
        dist.assign(n, INF);
        vector<bool> inqueue(n, false);

        queue<ll> q; q.push(src);
        dist[src] = 0;
        inqueue[src] = true;

        while (!q.empty()) {
            ll u = q.front(); q.pop();
            inqueue[u] = false;

            for (auto eid : g[u]) {
                auto const& e = edges[eid];
                if (e.flow >= e.cap) continue;
                if (dist[e.u] + e.cost < dist[e.v]) {
                    dist[e.v] = dist[e.u] + e.cost;
                    if (!inqueue[e.v]) {
                        q.push(e.v);
                        inqueue[e.v] = true;
                    }
                }
            }
        }
        return dist[sink] != INF;
    }

    ll min_cost = 0;
    ll dfs(ll u, ll f) {
        if (f == 0 or u == sink) return f;
        for (ll &i = ptr[u]; i < (ll)g[u].size();) {
            ll eid = g[u][i++];
            auto &e = edges[eid];
            if(e.flow >= e.cap or (dist[e.u] + e.cost) != dist[
                 e.v]) continue;
            ll newf = dfs(e.v, min(f, e.cap - e.flow));
            if (newf == 0) continue;
            e.flow += newf;
            edges[eid^1].flow -= newf;
            min_cost += e.cost * newf;
            return newf;
        }
        return 0;
    }

    ll max_flow = 0;
    pair<ll, ll> flow(bool reset_flow_cost = true) {
        if (reset_flow_cost) {
            max_flow = 0;
            min_cost = 0;
            for(ll u=0; u<n; u++) {
                for(auto eid : g[u]) {
                    auto &e = edges[eid];
                    e.flow = 0;
                }
            }
        }
        while (spfa()) {
            ptr.assign(n, 0);
            while (ll newf = dfs(src, INF))
                max_flow += newf;
        }
        return {min_cost, max_flow};
    }
};
```

6.3.2 Hungarian

Solves the **Assignment Problem:**

There are several standard formulations of the assignment problem (all of which are essentially equivalent). Here are some of them:

There are n jobs and n workers. Each worker specifies the amount of money they expect for a particular job. Each worker can be assigned to only one job. The objective is to assign jobs to workers in a way that minimizes the total cost.

Given an $n \times n$ matrix A, the task is to select one number from each row such that exactly one number is chosen from each column, and the sum of the selected numbers is minimized.

Given an $n \times n$ matrix A, the task is to find a permutation p of length n such that the value $\sum A[i][p[i]]$ is minimized.

Consider a complete bipartite graph with n vertices per part, where each edge is assigned a weight. The objective is to find a perfect matching with the minimum total weight.

It is important to note that all the above scenarios are "square" problems, meaning both dimensions are always equal to n. In practice, similar "rectangular" formulations are often encountered, where n is not equal to m, and the task is to select $\min(n, m)$ elements. However, it can be observed that a "rectangular" problem can always be transformed into a "square" problem by adding rows or columns with zero or infinite values, respectively.

We also note that by analogy with the search for a minimum solution, one can also pose the problem of finding a maximum solution. However, these two problems are equivalent to each other: it is enough to multiply all the weights by -1.

hungarian.cpp
Description: Solves the assignment problem
Time: $\mathcal{O}(n^3)$
06d970, 72 lines

```
// Hungaro
//
// Resolve o problema de assignment (matriz n x n)
// Colocar os valores da matriz em 'a' (pode < 0)
// assignment() retorna um par com o valor do
// assignment minimo, e a coluna escolhida por cada linha
// 0-idx
//
// O(n^3)

template<typename T> struct Hungarian {
  int n;
  vector<vector<T>> a;
  vector<T> u, v;
  vector<int> p, way;
  T inf;

  Hungarian(int n_) : n(n_), u(n+1), v(n+1), p(n+1), way(n+1) {
    a = vector<vector<T>>(n, vector<T>(n));
    inf = numeric_limits<T>::max();
  }

  pair<T, vector<int>> assignment() {
    for (int i = 1; i <= n; i++) {
      p[0] = i;
      int j0 = 0;
      vector<T> minv(n+1, inf);
      vector<int> used(n+1, 0);
      do {
        used[j0] = true;
        int i0 = p[j0], j1 = -1;
        T delta = inf;
        for (int j = 1; j <= n; j++) if (!used[j]) {
          T cur = a[i0-1][j-1] - u[i0] - v[j];
          if (cur < minv[j]) minv[j] = cur, way[j] = j0;
          if (minv[j] < delta) delta = minv[j], j1 = j;
        }
        for (int j = 0; j <= n; j++)
          if (used[j]) u[p[j]] += delta, v[j] -= delta;
          else minv[j] -= delta;
        j0 = j1;
      } while (p[j0] != 0);
      do {
        int j1 = way[j0];
        p[j0] = p[j1];
        j0 = j1;
      } while (j0);
    }
    vector<int> ans(n);
    for (int j = 1; j <= n; j++) ans[p[j]-1] = j-1;
    return make_pair(-v[0], ans);
  }
};

int32_t main(){ sws;
    ll n; cin >> n;
    Hungarian<ll> h(n);

    for(ll i=0; i<n; i++) {
        for(ll j=0; j<n; j++) {
            cin >> h.a[i][j];
        }
    }

    auto [cost, match] = h.assignment();

    cout << cost << endl;

    for(ll i=0; i<n; i++) {
        cout << i+1 << " " << match[i]+1 << endl;
    }
}
```

6.4 Coloring

Chromatic Number is the *minimum* number of colors to color all the vertices, so that no two adjacent vertices have the same color.

Chromatic Polynomial $P(K)$ is the number of ways to color a graph with K colors.

A **k-coloring** is the same as a partition of the vertex set into k independent sets, and the terms *k-partite* and *k-colorable* have the same meaning.

A *clique* of size N will force the Chromatic Number to be at least N. It is a *lower bound*.

The **maximum vertex degree** $(+1)$ in an undirected graph will be the *upper bound* on the Chromatic Number.

2-colorable graphs are exactly the bipartite graphs.

Four color theorem states that every planar graph can be 4-colored. A planar graph is a graph that can be drawn in a 2D plane without edges intersecting.

6.4.1 Compute the Chromatic Number

chromatic-number.cpp
Description: Using bitmask DP and Inclusion-Exclusion principle, compute the Chromatic Number of a small graph.
Time: $\mathcal{O}(2^n n)$
95be88, 32 lines

```
ll ChromaticNumber(const vector<vector<ll>> &g) {
    ll n = g.size();
    ll N = 1 << n;
    ll ans = n;
    // adjacency list using bitmask
    vector<ll> adj(n);
    for (ll u = 0; u < n; u++)
        for (auto v : g[u])
            adj[u] |= (1 << v);

    // choose some primes to avoid hacking
    for (ll d : {7}) { //,11,21,33,87,93}) {
        ll mod = 1e9; mod += d;
        vector<ll> ind(N), aux(N, 1);
        ind[0] = 1;
        for (ll s = 1; s < N; s++) {
            ll u = __builtin_ctzll(s);
            ind[s] = ind[s ^ (1 << u)] + ind[(s ^ (1 << u)) & ~
                 adj[u]];
        }
        for (ll k = 1; k < ans; k++) {
            ll w = 0;
            for (ll i = 0; i < N; ++i) {
                ll s = i ^ (i >> 1); // gray-code
                aux[s] = (aux[s] * ind[s]) % mod;
                w += (i & 1) ? aux[s] : -aux[s];
            }
            if (w % mod) ans = min(ans, k);
        }
    }
    return ans;
}
```

6.5 Shortest Paths

For weighted directed graphs

6.5.1 Dijkstra

Single Source and there **cannot** be any negative weighted edges.

dijkstra.cpp
Description: By keeping track of the distances sorted using an priority queue of candidates. if an edge can reduce the current min distance, insert into the priority queue. ONLY when the vertice is dequeued and its cost is <= d[u], it is in fact a part of a shortest path
Time: $\mathcal{O}((V+E)\log V)$
76e3a5, 21 lines

```
priority_queue<pll, vector<pll>, greater<pll>> pq; // min pq
vector<vector<pll>> g(MAX);
vector<ll> d(MAX, INF);

void dijkstra(ll start){
    pq.push({0, start});
    d[start] = 0;

    while(!pq.empty()){

        auto [cost, u] = pq.top(); pq.pop();
        if (cost > d[u]) continue;

        for (auto [v, w] : g[u]) {
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
                pq.push({d[v], v});
            }
        }
    }
}
```

By inverting the sorting order, Dijkstra can be modified for the opposite operation: *longest paths*.

Furthermore, Dijkstra be extended to keep track of more information, such as:

- how many minimum-price routes are there? (modulo $10^9 + 7$)
- what is the minimum number of flights in a minimum-price route?
- what is the maximum number of flights in a minimum-price route?

extendedDijkstra.cpp
Description: Also counts the numbers of shortest paths, the minimum and maximum number of edges transversed in any shortest path.
Time: $\mathcal{O}((V+E)\log V)$
f93094, 32 lines

```
priority_queue<pll, vector<pll>, greater<pll>> pq; // min pq
vector<vector<pll>> g(MAX);
vector<ll> d(MAX, INF), ways(MAX, 0), mx(MAX, -INF), mn(MAX,
     INF);
// INF = INT64_MAX

void dijkstra(ll start){
    pq.push({0, start});
    ways[start] = 1;
    d[start] = mn[start] = mx[start] = 0;

    while( !pq.empty() ){
        auto [p1, u] = pq.top(); pq.pop();
        if (p1 > d[u]) continue;
        for(auto [v, p2] : g[u]){
            // reset info, shorter path found, previous ones
                 are discarted
            if (d[u] + p2 < d[v]){
                d[v] = d[u] + p2;
                ways[v] = ways[u];
                mx[v] = mx[u]+1;
                mn[v] = mn[u]+1;

                pq.push({d[v], v});
            }
            // same distance, different path, update info
            else if (d[u] + p2 == d[v]) {
                ways[v] = (ways[v] + ways[u]) % MOD;
                mn[v] = min(mn[v], mn[u]+1);
                mx[v] = max(mx[v], mx[u]+1);
            }
        }
    }
}
```

6.5.2 Bellman-Ford

Single Source and it **supports** negative edges

Conjecture: After at most n-1 (Vertices-1) iterations, all shortest paths will be found.

bellman-ford.cpp
Description: n-1 iterations is sufficient to find all shortest paths
Time: $\mathcal{O}(V*E)$ -> $\mathcal{O}(N^2)$
d749f1, 15 lines

```
using T = array<ll, 3>;
vector<T> edges;
vector<ll> d(MAX, INF);
// INF = 0x3f3f3f3f3f3f3f3f, to avoid overflow
void BellmanFord(ll src, ll n) {
    d[src] = 0;
    for(ll i=0; i<n-1; i++) { // n-1 iterations
        for(auto [u, v, w] : edges) {
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
            }
        }
    }
}
```

By iterating once more, one can check if the last iteration reduced once again any distance. If so, it means that there must be a negative cycle, because the shortest distance should have been found before elseway.

To retrieve the negative cycle itself, one can keep track of the last vertice that reaches a considered vertice

bellman-ford-cycle.cpp
Description: By using the property that n-1 iterations is sufficient to find all shortest paths in a graph that doesn't have negative cycles. Iterate n times and retrieve the path using a vector of parents
Time: $\mathcal{O}(V*E)$ -> $\mathcal{O}(N^2)$
0506b5, 35 lines

```
using T = array<ll, 3>;
vector<T> edges;
vector<ll> d(MAX, INF), p(MAX, -1);
vector<ll> cycle;
// INF = 0x3f3f3f3f3f3f3f3f, to avoid overflow

void BellmanFordCycle(ll src, ll n) {
    d[src] = 0;
    ll x = -1; // possible node inside a negative cycle
    for(ll i=0; i<n; i++) { // n iterations
        x = -1;
        for(auto [u, v, w] : edges) {
            if (d[u] + w < d[v]) {
                d[v] = d[u] + w;
                p[v] = u;
                x = v;
            }
        }
    }

    if (x != -1) {
        // set x to a node, contained in a cycle in p[]
        for(ll i=0; i<n; i++) x = p[x];

        ll tmp = x;
        do {
            cycle.pb(tmp);
            tmp = p[tmp];
        }
        while (tmp != x);
        cycle.pb(x);

        reverse(cycle.begin(), cycle.end());
    }
}
```

s

6.5.3 Floyd Warshall

All-Pair Shortest Paths

floyd-warshall.cpp
Description: By using an auxiliar vertice, check if a smaller path exists between a pair (u, v) of vertices, if so, update minimum distance.
Time: $\mathcal{O}(V^3)$
fa5f60, 15 lines

```
// N < sqr3(1e8) = 460
ll N = 200;

// d[u][v] = INF (no edge)
vector<vll> d(N+1, vll(N+1, INF));

void floydWarshall() { // O(N^3)
    for(ll i=1; i<=N; i++) d[i][i] = 0;

    for(ll aux=1; aux<=N; aux++)
        for(ll u=1; u<=N; u++)
            for(ll v=1; v<=N; v++)
                if (d[u][aux] < INF and d[v][aux] < INF)
                    d[u][v] = min(d[u][v], d[u][aux] + d[v][aux
                        ]);
}
```

6.6 Eulerian Path

An *Eulerian trail (or Eulerian path)* is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices). Similarly, an *Eulerian circuit or Eulerian cycle* is an Eulerian trail that starts and ends on the same vertex.

Euler's Theorem: A connected graph has an Euler cycle if and only if every vertex has even degree.

An **Eulerian Graph** is a graph with an eulerian circuit, which implies that it is connected and also all vertices have even degree.

What conditions are required for a valid Eulerian Path/Circuit?

That depends on what kind of graph you're dealing with. Altogether there are four flavors of the Euler path/circuit problem we care about:

	Eulerian Circuit	Eulerian Path
Undirected Graph	Every vertex has an even degree.	Either every vertex has even degree or exactly two vertices have odd degree.
Directed Graph	Every vertex has equal indegree and outdegree	At most one vertex has (outdegree) - (indegree) = 1 and at most one vertex has (indegree) - (outdegree) = 1 and all other vertices have equal in and out degrees.

If there are extra edges, which ones can be removed to generate a maximum size euler path?

Answer: In an undirected graph, one can remove the ones connecting two *odd* degree nodes, turning then into *even* ones. Do this until there are less or equal than 2 *odd* degree nodes.

hierholzer.cpp
Description: Check existence conditions and produce path if possible
Time: $\mathcal{O}(E)$
ced657, 110 lines

```
// Euler Circuit - Undirected Graph (1-idx)
pair<bool, vector<ll>> circuit(ll n, vector<pll> &edges, ll src
    = 1) {
    ll m = edges.size();
    vector<vector<pll>> g(n+1);

    for(ll i=0; i<m; i++) {
        auto [a, b] = edges[i];
        g[a].pb({b, i});
        g[b].pb({a, i});
    }

    // check for even degree
    for(ll i=0; i<=n; i++) {
        if (g[i].size() % 2) return {false, {}};
    }

    vector<ll> ans;
    vector<bool> vis(m, 0);
    function<void (ll)> dfs = [&](ll u) {
        while(g[u].size()) {
            auto [v, idx] = g[u].back();
            g[u].pop_back();
            if (vis[idx]) continue;
            vis[idx] = 1;
            dfs(v);
        }
        ans.pb(u);
    };
    dfs(src);

    // check for connectivity
    if (ans.size() != m + 1) return {false, {}};

    return {true, ans};
}

// Euler Path - Undirected Graph (1-idx)
pair<bool, vector<ll>> path(ll n, vector<pll> &edges, ll src,
    ll dst) {
    ll m = edges.size();
    vector<vector<pll>> g(n+1);

    for(ll i=0; i<m; i++) {
        auto [a, b] = edges[i];
        g[a].pb({b, i});
        g[b].pb({a, i});
    }

    // check for even degree (except src and dst)
    for(ll i=0; i<=n; i++) {
        if (i == src or i == dst) continue;
        if (g[i].size() % 2) return {false, {}};
    }

    vector<ll> ans;
    vector<bool> vis(m, 0);
    function<void (ll)> dfs = [&](ll u) {
        while(g[u].size()) {
            auto [v, idx] = g[u].back();
            g[u].pop_back();
            if (vis[idx]) continue;
            vis[idx] = 1;
            dfs(v);
        }
        ans.pb(u);
    };
    dfs(src);

    // check for connectivity
    if (ans.size() != m + 1) return {false, {}};

    reverse(ans.begin(), ans.end());
    return {true, ans};
}

// Euler Path - Directed Graph (1-idx)
pair<bool, vector<ll>> path(ll n, vector<pll> &edges, ll src,
    ll dst) {
    ll m = edges.size();
    vector<vector<ll>> g(n+1);

    vector<ll> in(n+1, 0), out(n+1, 0);
    for(auto [a, b] : edges) {
        g[a].pb(b);
        out[a]++, in[b]++;
    }

    // check for degrees
    for(ll i=0; i<=n; i++) {
        if (i == src or i == dst) continue;
        if (in[i] - out[i] != 0) return {false, {}};
    }
    if (out[src] != in[src] + 1) return {false, {}};
    if (in[dst] != out[dst] + 1) return {false, {}};

    vector<ll> ans;
    function<void (ll)> dfs = [&](ll u) {
        while(g[u].size()) {
            auto v = g[u].back();
            g[u].pop_back();
            dfs(v);
        }
        ans.pb(u);
    };
    dfs(src);

    // check for connectivity
    if (ans.size() != m + 1) return {false, {}};

    reverse(ans.begin(), ans.end());
    return {true, ans};
}
```

6.7 Undirected Graph

Bridges and Articulation Points are concepts for undirected graphs!

6.7.1 Bridges (Cut Edges)

Also called **isthmus** or **cut arc**.

A back-edge is never a bridge!

A **lowlink** for a vertice U is the closest vertice to the root reachable using only span edges and a *single* back-edge, starting in the subtree of U.

After constructing a DFS Tree, an edge (u, v) is a bridge $\iff$ there is no back-edge from v (or a descendent of v) to u (or an ancestor of u)

To do this efficiently, it's used $tin[i]$ (entry time of node i) and $low[i]$ (minimum entry time considering all nodes that can be reached from node i).

In another words, a edge (u, v) is a bridge $\iff$ the low[v] ¿ tin[u].

bridges.cpp
Description: Using the concepts of entry time (tin) and lowlink (low), an edge is a bridge if, and only if, low[v] > tin[u]
Time: $\mathcal{O}(V + E)$
87e0d3, 25 lines

```
vector<vll> g(MAX);
ll timer = 1;
ll tin[MAX], low[MAX];
vector<pll> bridges;

void dfs(ll u, ll p = -1){
    tin[u] = low[u] = timer++;
    for(auto v : g[u]) if (v != p) {
        if (tin[v]) // v was visited ({u,v} is a back-edge)
            // considering a single back-edge:
            low[u] = min(low[u], tin[v]);
        else { // v wasn't visited ({u, v} is a span-edge)
            dfs(v, u);
            // after low[v] was computed by dfs(v, u):
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u])
                bridges.pb({u, v});
        }
    }
}

void findBridges(ll n) {
    for(ll i=1; i<=n; i++) if (!tin[i])
        dfs(i);
}
```

6.7.2 Bridge Tree

After merging *vertices* of a **2-edge connected component** into single vertices, and leaving only bridges, one can generate a Bridge Tree.

Every **2-edge connected component** has following properties:

- For each pair of vertices A, B inside the same component, there are at least 2 distinct paths from A to B (which may repeat vertices).

bridgeTree.cpp
Description: After finding bridges, set an component id for each vertice, then merge vertices that are in the same 2-edge connected component
Time: $\mathcal{O}(V + E)$
6d00bd, 47 lines

```
// g: u -> {v, edge id}
vector<vector<pll>> g(MAX);
vector<vll> gc(MAX);
ll timer = 1;
ll tin[MAX], low[MAX], comp[MAX];
bool isBridge[MAX];
```

```
void dfs(ll u, ll p = -1) {
    tin[u] = low[u] = timer++;
    for(auto [v, id] : g[u]) if (v != p) {
        if (tin[v])
            low[u] = min(low[u], tin[v]);
        else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] > tin[u])
                isBridge[id] = 1;
        }
    }
}

void dfs2(ll u, ll c, ll p = -1) {
    comp[u] = c;
    for(auto [v, id] : g[u]) if (v != p) {
        if (isBridge[id]) continue;
        if (!comp[v]) dfs2(v, c, u);
    }
}

void bridgeTree(ll n) {
    // find bridges
    for(ll i=1; i<=n; i++) if (!tin[i])
        dfs(i);

    // find components
    for(ll i=1; i<=n; i++) if (!comp[i])
        dfs2(i, i);

    // condensate into a TREE (or TREES if disconnected)
    for(ll u=1; u<=n; u++) {
        for(auto [v, id] : g[u]) {
            if (comp[u] != comp[v]) {
                gc[comp[u]].pb(comp[v]);
            }
        }
    }
}
```

6.7.3 Articulation Points

One Vertice in a graph is considered a Articulation Points or Cut Vertice if its removal in the graph will generate more disconnected components

articulation.cpp

Description: if low[v] >= tin[u], u is an articulation points The root is a corner case

Time: $\mathcal{O}(V+E)$

8707a0, 29 lines

```
vector<vll> g(MAX);
ll timer = 1;
ll low[MAX], tin[MAX], isAP[MAX];
// when vertex i is removed from graph
// isAP[i] is the quantity of new disjoint components created
// isAP[i] >= 1 {i is a Articulation Point}
void dfs(ll u, ll p = -1) {
    low[u] = tin[u] = timer++;

    for(auto v : g[u]) if (v != p) {
        if (tin[v]) // visited
            low[u] = min(low[u], tin[v]);
        else { // not visited
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= tin[u])
                isAP[u]++;
        }
    }

    // corner case: root
    if (p == -1 and isAP[u]) isAP[u]--;
}

void findAP(ll n) {
    for(ll i=1; i<=n; i++) if (!tin[i])
        dfs(i);
}
```

6.7.4 Block Cut Tree

After merging *edges* of a **2-vertex connected component** into single vertices, one can obtain a block cut tree.

2-vertex connected components are also called as biconnected component

Every bridge by itself is a biconnected component

Each edge in the block-cut tree connects exactly an Articulation Point and a biconnected component (bipartite graph)

Each biconnected component has the following properties:

- For each pair of edges, there is a cycle that contains both edges
- For each pair of vertices A, B inside the same connected component, there are at least 2 distinct paths from A to B (which do not repeat vertices).

blockCutTree.cpp

Description: After Merging 2-Vertex Connected Components, one can generate a block cut tree

Time: $\mathcal{O}(V+E)$

f752d5, 100 lines

```
// Block-Cut Tree (bruno monteiro)
//
// Cria a block-cut tree, uma arvore com os blocos
// e os pontos de articulacao
// Blocos sao as componentes 2-vertice-conexos maximais
// Uma 2-coloracao da arvore eh tal que uma cor sao
// os componentes, e a outra cor sao os pontos de articulacao
//
// Funciona para grafo nao conexo
//
// isAP[i] responde o numero de novas componentes conexas
// criadas apos a remocao de i do grafo g
// Se isAP[i] >= 1, i eh ponto de articulacao
//
// Para todo i < blocks.size()
// blocks[i] eh uma componente 2-vertce-conexa maximal
// blockEdges[i] sao as arestas do bloco i
//
// tree eh a arvore block-cut-tree
// tree[i] eh um vertice da arvore que corresponde ao bloco i
//
// comp[i] responde a qual vertice da arvore vertice i pertence
//
// Arvore tem no maximo 2n vertices
//
// O(n+m)

// 0-idx graph!!!
vector<vll> g(MAX), tree, blocks;
vector<vector<pll>> blockEdges;
stack<ll> st; // st for vertices,
stack<pll> st2; // st2 for edges
vector<ll> low, tin, comp, isAP;
ll timer = 1;

void dfs(ll u, ll p = -1) {
    low[u] = tin[u] = timer++;

    st.push(u);

    // add only back-edges to stack
    if (p != -1) st2.push({u, p});
    for(auto v : g[u]) if (v != p) {
        if (tin[v] != -1) // visited
            st2.push({u, v});
    }

    for(auto v : g[u]) if (v != p) {
        if (tin[v] != -1) // visited
            low[u] = min(low[u], tin[v]);
        else { // not visited
            dfs(v, u);
            low[u] = min(low[u], low[v]);

            if (low[v] >= tin[u]) {
                isAP[u] += 1;

                blocks.pb(vll(1, u));
                while(blocks.back().back() != v)
                    blocks.back().pb(st.top()), st.pop();

                blockEdges.pb(vector<pll>(1, st2.top())), st2.
                    pop();
                while(blockEdges.back().back() != pair<ll, ll>(
                    v, u))
                    blockEdges.back().pb(st2.top()), st2.pop();
            }
        }
    }

    // corner case: root
    if (p == -1 and isAP[u]) isAP[u]--;
}

void blockCutTree(ll n) {

    // initialize vectors and reset
    tree.clear(), blocks.clear(), blockEdges.clear();
    st = stack<ll>(), st2 = stack<pll>();
    tin.assign(n, -1);
    low.assign(n, 0), comp.assign(n, 0), isAP.assign(n, 0);
    timer = 1;

    // find Articulation Points
    for(ll i=0; i<n; i++) if (tin[i] == -1)
        dfs(i);

    // set component id for APs
    tree.assign(blocks.size(), vll());
    for(ll i=0; i<n; i++) if (isAP[i])
        comp[i] = tree.size(), tree.pb(vll());

    // set component id for non-APs and construct tree
    for(ll u=0; u<(ll)blocks.size(); u++) {
        for(auto v : blocks[u]) {
```

```
            if (!isAP[v])
                comp[v] = u;
            else
                tree[u].pb(comp[v]), tree[comp[v]].pb(u);
        }
    }
}
```

6.7.5 Strong Orientation

A **strong orientation** of an undirected graph is an assignment of a direction to each edge that makes it a strongly connected graph. That is, after the orientation we should be able to visit any vertex from any vertex by following the directed edges.

Of course, this cannot be done to every graph. Consider a **bridge** in a graph. We have to assign a direction to it and by doing so we make this bridge "crossable" in only one direction. That means we can't go from one of the bridge's ends to the other, so we can't make the graph strongly connected.

Now consider a DFS through a bridgeless connected graph. Clearly, we will visit each vertex. And since there are no bridges, we can remove any DFS tree edge and still be able to go from below the edge to above the edge by using a path that contains at least one back edge. From this follows that from any vertex we can go to the root of the DFS tree. Also, from the root of the DFS tree we can visit any vertex we choose. We found a strong orientation!

In other words, to strongly orient a bridgeless connected graph, run a DFS on it and let the DFS tree edges point away from the DFS root and all other edges from the descendant to the ancestor in the DFS tree.

The result that bridgeless connected graphs are exactly the graphs that have strong orientations is called **Robbins' theorem**.

Acyclic Graph Orientation

Problem: Given an undirected graph, your task is to choose a direction for each edge so that the resulting directed graph is acyclic.

Solution: Do a dfs tree, every span-edge is oriented according to the dfs transversal, and every back-edge is oriented contrary to the dfs transversal

6.8 Directed Graph

6.8.1 Topological Sort

Sort a directed graph with no cycles (DAG) in an order which each source of an edge is visited before the sink of this edge.

Cannot have cycles, because it would create a contradition of which vertices whould come before.

It can be done with a DFS, appending in the reverse order of transversal. Also a stack can be used to reverse order

toposort.cpp
Description: Using DFS pos order transversal and inverting the order, one can obtain the topological order
Time: $\mathcal{O}(V+E)$
75f781, 17 lines

```
vector<vll> g(MAX, vll());
vector<bool> vis;
vll topological;

void dfs(ll u) {
    vis[u] = 1;
    for(auto v : g[u]) if (!vis[v]) dfs(v);
    topological.pb(u);
}

// 1 - indexed
void topological_sort(ll n) {
    vis.assign(n+1, 0);
    topological.clear();
    for(ll i=1; i<=n; i++) if (!vis[i]) dfs(i);
    reverse(topological.begin(), topological.end());
}
```

s

6.8.2 Kosaraju

A Strongly Connected Component is a maximal subgraph in which every vertex is reachable from any vertex inside this same subgraph.

A important *property* is that the inverted graph or transposed graph has the same SCCs as the original graph.

kosaraju.cpp
Description: By using the fact that the inverted graph has the same SCCs, just do a DFS twice to find all SCCs. A condensated graph can be created if wished. The condensated graph is a DAG!!
Time: $\mathcal{O}(V+E)$
2eb446, 43 lines

```
struct Kosaraju {
    ll n;
    vector<vll> g, gi, gc;
    vector<bool> vis;
    vector<ll> comp;
    stack<ll, vll> st;

    void dfs(ll u) { // g
        vis[u] = 1;
        for(auto v : g[u]) if (!vis[v]) dfs(v);
        st.push(u);
    }

    void dfs2(ll u, ll c) { // gi
        comp[u] = c;
        for(auto v : gi[u]) if (comp[v] == -1) dfs2(v, c);
    }

    Kosaraju(vector<vll> &g_) : n(g_.size()-1), g(g_),
        gi(n+1), gc(n+1), vis(n+1, 0), comp(n+1, -1), st() { //
            1-idx

        for(ll u=1; u<=n; u++) {
            for(auto v : g[u]) {
                gi[v].pb(u);
            }
        }

        for(ll i=1; i<=n; i++) if (!vis[i]) dfs(i);

        while(!st.empty()) {
            auto u = st.top(); st.pop();
            if (comp[u] == -1) dfs2(u, u);
        }

        for(ll u=1; u<=n; u++) {
            for(auto v : g[u]) {
                if (comp[u] != comp[v]) {
                    gc[comp[u]].pb(comp[v]);
                }
            }
        }
    }
};
```

6.8.3 2-SAT

SAT (Boolean satisfiability problem) is NP-Complete.

2-SAT is a restriction of the SAT problem, in 2-SAT every clause has exactly two variables: $(X_1 \vee X_2) \wedge (X_2 \vee X_3)$

Every restriction or implication are represented in the graph as directed edges.

The algorithm uses kosaraju to check if any (X and $\neg X$) are in the same Strongly Connected Component (which implies that the problem is impossible).

If it doesn't, there is at least one solution, which can be generated using the topological sort of the same kosaraju (opting for the variables that appers latter in the sorted order)

2sat.cpp
Description: Kosaraju to find if there are SCCs. If there are not cycles, use toposort to choose states
Time: $\mathcal{O}(V+E)$
87417c, 83 lines

```
// 0-idx graph !!!!
struct TwoSat {
    ll N; // needs to be the twice of the number of variables
    // node with idx 2x => variable x
    // node with idx 2x+1 => variable !x

    vector<vll> g, gi;
    // g = graph; gi = transposed graph (all edges are inverted
        )

    TwoSat(ll n) { // number of variables (add +1 faor 1-idx)
        N = 2*n;
        g.assign(N, vll());
        gi.assign(N, vll());
    }

    ll idx; // component idx
    vector<ll> comp, order; // topological order (reversed)
    vector<bool> vis, chosen;
    //  chosen[x] == 0 -> x was assigned
    //  chosen[x] == 1 -> !x was assigned

    // dfs and dfs2 are part of kosaraju algorithm
    void dfs(ll u) {
        vis[u] = 1;
        for (ll v : g[u]) if (!vis[v]) dfs(v);
        order.pb(u);
    }

    void dfs2(ll u, ll c) {
        comp[u] = c;
```

```
        for (ll v : gi[u]) if (comp[v] == -1) dfs2(v, c);
    }

    bool solve() {
        vis.assign(N, 0);
        order = vector<ll>();
        for (ll i = 0; i < N; i++) if (!vis[i]) dfs(i);

        comp.assign(N, -1); // comp = 0 can exist
        idx = 1;
        for(ll i=(ll)order.size()-1; i>=0; i--) {
            ll u = order[i];
            if (comp[u] == -1) dfs2(u, idx++);
        }

        chosen.assign(N/2, 0);
        for (ll i = 0; i < N; i += 2) {
            // x and !x in the same component => contradiction
            if (comp[i] == comp[i+1]) return false;
            chosen[i/2] = comp[i] < comp[i+1]; // choose latter
                node
        }
        return true;
    }

    // a (with flagA) implies => b (with flagB)
    void add(ll a, bool fa, ll b, bool fb) {
        // {fa == 0} => a
        // {fa == 1} => !a
        a = 2*a + fa;
        b = 2*b + fb;
        g[a].pb(b);
        gi[b].pb(a);
    }

    // force a state for a certain variable (must be true)
    void force(ll a, bool fa) {
        add(a, fa^1, a, fa);
    }

    // xor operation: one must exist, and only one can exist
    void exclusive(ll a, bool fa, ll b, bool fb) {
        add(a, fa^0, b, fb^1);
        add(a, fa^1, b, fb^0);
        add(b, fb^0, a, fa^1);
        add(b, fb^1, a, fa^0);
    }

    // nand operation: no more than one can exist
    void nand(ll a, bool fa, ll b, bool fb) {
        add(a, fa^0, b, fb^1);
        add(b, fb^0, a, fa^1);
    }
};
```

6.9 Minimum Spanning Tree

6.9.1 Undirected Minimum Spanning Tree

A **minimum spanning tree (MST)** or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

kruskal.cpp

Description: Sort all edges in crescent order by weight, include all edges which joins two disconnected trees. In case of tie, choose whichever. Dont include edges that will join a already connected part of the tree.

Time: $\mathcal{O}(E \log E\alpha)$

206ba3, 21 lines

```
// use DSU struct
struct DSU{};

set<array<ll, 3>> edges;

int32_t main(){ sws;
    ll n, m; cin >> n >> m;
    DSU dsu(n+1);
    for(ll i=0; i<m; i++) {
        ll u, v, w; cin >> u >> v >> w;
        edges.insert({w, u, v});
    }
    ll minCost = 0;
    for(auto [w, u, v] : edges) {
        if (dsu.find(u) != dsu.find(v)) {
            dsu.join(u, v);
            minCost += w;
        }
    }
    cout << minCost << endl;
}
```

6.9.2 Directed Minimum Spanning Tree

A **spanning arborescence of minimum weight**, sometimes called an **optimum branching**, is the directed analog of the *minimum spanning tree problem.*

Edmonds' algorithm with Tarjan's optimization

directed-mst.cpp

Description: Given a vector of directed edges with weights and a root, compute the cost of the directed minimum spanning tree (all nodes reachable from root); or return INF if it doesn't exist.

Time: $\mathcal{O}(m \log(n))$ for sparse graphs, $\mathcal{O}(n^2)$ for dense graphs;

3ad7a5, 93 lines

```
using T = array<ll, 3>;

struct directedMST { // 0-idx
    struct Node {
        pair<ll, ll> val;
        ll lazy = 0;
        Node *l = NULL, *r = NULL;

        void push() {
            val.ff += lazy;
            if (l) l->lazy += lazy;
            if (r) r->lazy += lazy;
            lazy = 0;
        }
    };

    void erase(Node *i) {
        if (i) {
            erase(i->l), erase(i->r);
            delete i;
        }
    }

    pair<ll, ll> pop(Node* &i) {
        i->push();
        auto ret = i->val;
        Node *tmp = i;
        join(i->l, i->r);
        i = i->l;
        if (i) i->lazy -= ret.ff;
        delete tmp;
        return ret;
    }

    ll n, root;
    vector<T> edges;
    vector<ll> group;
    vector<Node*> h;

    ll find(ll i) {
        return (group[i] == i) ? i : (group[i] = find(group[i])
            );
    }

    void join(Node* &a, Node* b) {
        if (!a) swap(a, b);
        if (!b) return;
        a->push(), b->push();
        if (a->val > b->val) swap(a, b);
        join((rand() % 2) ? a->l : a->r, b);
    }

    // 0-idx [0, n), edges = {u, v, w}, from U, to V, with
        weight W
    directedMST(ll n_, ll root_, vector<T> &edges_)
        : n(n_), root(root_), edges(edges_), group(n), h(n) {
        iota(group.begin(), group.end(), 0);
    }

    ~directedMST() {
        for(auto i : h) erase(i);
    };

    ll solve() {
        for(auto [u, v, w] : edges) {
            join(h[v], new Node({{w, u} }));
        }

        vector<ll> p(n, -1), path(n);
        p[root] = root;
        ll ans = 0;

        for(ll i=0; i<n; i++) {
            ll u = i, at = 0;
            while(p[u] == -1) {
                if (!h[u]) return INF;

                path[at++] = u, p[u] = i;
                auto [mn, v] = pop(h[u]);
                ans += mn;

                if (p[u = find(v)] == i) {
                    while(find(v = path[--at]) != u) {
                        join(h[u], h[v]);
                        h[v] = NULL;
                        group[find(v)] = u;
                    }
                    p[u] = -1;
                }
            }
        }

        return ans;
    }
};
```

6.10 Trees

6.10.1 Centroid

A **centroid** of a tree with size N is defined as a node such that, when the tree is rooted at it, no other nodes have a subtree of size greater than $N/2$.

We can find a centroid in a tree by starting at any node and at, each step, loop through all of its children. If all of its children have a subtree whose size is $\leq N/2$, then this node is a centroid. Otherwise, move to the child with a subtree whose size is $> N/2$ and repeat until you find a centroid.

A tree can have a single centroid or two centroids.

centroid.cpp
Description: find the centroid(s) of a tree
Time: $\mathcal{O}(V)$
66a9c6, 21 lines

```
// 0-idx
struct Centroid {
    vector<vector<ll>> g;
    ll n;
    vector<ll> sz, cs; // cs has the vertices that are
        centroids

    void find_centroid(ll u, ll p = -1) {
        sz[u] = 1;
        bool cent = true;
        for(auto v : g[u]) if (v != p) {
            find_centroid(v, u), sz[u] += sz[v];
            if (sz[v] > n/2) cent = false;
        }
        if (cent and n - sz[u] <= n/2) cs.pb(u);
    }

    // initialize G with the correct size, so that n = vertices
        (exactly) !!
    Centroid(vector<vector<ll>> &g_) : g(g_), n(g.size()), sz(n
        ) { find_centroid(0); }
};
```

6.10.2 Centroid Decomposition

After finding a centroid, and removing it from the tree, several subtrees will be formed. By applying this function several times until no more splitting can be done, we are doing a centroid decomposition.

Centroid Tree

By ordering the nodes by the time it was removed, we can create the centroid tree.

- 1. The centroid tree has height of $O(\log n)$;
- 2. A vertex belongs to the component (original tree) of *all* its ancestors (centroid tree).
- 3. The path from a to b can be decomposed into the path from a to $lca(a,b)$ and the path from $lca(a,b)$ to b. **Note** that the distance between nodes it's still measured in reference to the original tree.
- 4. Each one of the n^2 paths of the original tree is the concatenation of two paths in a set of $O(n \log(n))$ paths from a node to all its ancestors in the centroid decomposition.

6.10.3 Tree Isomorphism

isomorphism.cpp
Description: compute the hash value of a tree (root and unrooted)
Time: $\mathcal{O}(V \log V)$
fd7294, 39 lines

```
map<vector<int>, int> hashmap;

// 0-idx implementation
struct Tree {
    vector<vector<int>> g;
    int n;
    vector<int> sz, cs;

    // initialize G with the correct size, so that n = vertices
        (exactly) !!
    Tree(vector<vector<int>> &g_) : g(g_), n(g.size()), sz(n) {
        }

    // function to get the centroid(s) of a tree and appends to
        a vector
    void find_centroid(int u, int p = -1) {
        sz[u] = 1;
        bool cent = true;
        for(auto v : g[u]) if (v != p) {
            find_centroid(v, u), sz[u] += sz[v];
            if (sz[v] > n/2) cent = false;
        }
        if (cent and n - sz[u] <= n/2) cs.pb(u);
    }

    // get the hash of a rooted tree in (root = u) (returned
        hash is int)
    int hash(int u, int p = -1) {
        vector<int> h;
        for(int v : g[u]) if (v != p) h.pb(hash(v, u));
        sort(h.begin(), h.end());
        if (!hashmap.count(h)) hashmap[h] = hashmap.size();
        return hashmap[h];
    }

    // get the hash of an unrooted tree (returned hash is long
        long)
    ll uhash() {
        find_centroid(0); // 0-idx
        if (cs.size() == 1) return hash(cs[0]);
        ll h1 = hash(cs[0], cs[1]), h2 = hash(cs[1], cs[0]);
        return (min(h1, h2) << 30) + max(h1, h2);
    }
};
```

6.10.4 Lowest Common Ancestor (LCA)

Besides binary lifting, there is also the method of creating a tree-transversing array (euler tour), and then performing a range minimum query in the $tin[a]$ and $tin[b]$ to find $lca(a,b)$ using sparse tables.

lca.cpp
Description: Solves LCA for trees
Time: $\mathcal{O}(N \log(N))$ to build, $\mathcal{O}(\log(N))$ per query
7afc1a, 54 lines

```
struct BinaryLifting {
    ll n, logN = 20; // ~1e6
    vector<vll> g;
    vector<ll> depth;
    vector<vll> up;

    BinaryLifting(vector<vll> &g_)
     : g(g_), n(g_.size() + 1) { // 1-idx
        depth.assign(n, 0);

        while((1 << logN) < n) logN++;
        up.assign(n, vll(logN, 0));
        build();
    }

    void build(ll u = 1, ll p = -1) {
        for(ll i=1; i<logN; i++) {
            up[u][i] = up[ up[u][i-1] ][i-1];
        }

        for(auto v : g[u]) if (v != p) {
            up[v][0] = u;
            depth[v] = depth[u] + 1;
            build(v, u);
        }
    }

    ll go(ll u, ll dist) { // O(log(n))
        for(ll i=logN-1; i>=0; i--) { // bigger jumps first
            if (dist & (1LL << i)) {
                u = up[u][i];
            }
        }
        return u;
    }

    ll lca(ll a, ll b) { // O(log(n))
        if (depth[a] < depth[b]) swap(a, b);
        a = go(a, depth[a] - depth[b]);
        if (a == b) return a;

        for(ll i=logN-1; i>=0; i--) {
            if (up[a][i] != up[b][i]) {
                a = up[a][i];
                b = up[b][i];
            }
        }
        return up[a][0];
    }

    ll lca(ll a, ll b, ll root) { // lca(a, b) when tree is
        rooted at 'root'
        return lca(a, b)^lca(b, root)^lca(a, root); // magic
    }
};
```

binary-lifting.cpp
Description: Binary Lifting to compute the min, max edge weight present in the simple path a, lca(a, b), b
Time: $\mathcal{O}(N \log(N))$ to build; $\mathcal{O}(\log(N))$ per query
75ba37, 67 lines

```
struct BinaryLifting {
    ll n, logN = 20; // ~1e6
    vector<vpll> g;
    vector<ll> depth;
    vector<vll> up, mx, mn;
```

```
    BinaryLifting(vector<vpll> &g_)
     : g(g_), n(g_.size() + 1) { // 1-idx
        depth.assign(n, 0);

        while((1 << logN) < n) logN++;
        up.assign(n, vll(logN, 0));
        mx.assign(n, vll(logN, -INF));
        mn.assign(n, vll(logN, INF));
        build();
    }

    void build(ll u = 1, ll p = -1) {

        for(ll i=1; i<logN; i++) {
            mx[u][i] = max(mx[u][i-1], mx[ up[u][i-1] ][i-1]);
            mn[u][i] = min(mn[u][i-1], mn[ up[u][i-1] ][i-1]);
            up[u][i] = up[ up[u][i-1] ][i-1];
        }

        for(auto [v, w] : g[u]) if (v != p) {
            mx[v][0] = mn[v][0] = w;
            up[v][0] = u;
            depth[v] = depth[u] + 1;
            build(v, u);
        }
    }

    array<ll, 3> go(ll u, ll dist) { // O(log(n))
        ll mxval = -INF, mnval = INF;
        for(ll i=logN-1; i>=0; i--) { // bigger jumps first
            if (dist & (1LL << i)) {
                mxval = max(mxval, mx[u][i]);
                mnval = min(mnval, mn[u][i]);
                u = up[u][i];
            }
        }
        return {u, mxval, mnval};
    }

    array<ll, 3> query(ll u, ll v) { // O(log(n))
        if (depth[u] < depth[v]) swap(u, v);

        auto [a, mxval, mnval] = go(u, depth[u] - depth[v]);
        ll b = v;

        if (a == b) return {a, mxval, mnval};

        for(ll i=logN-1; i>=0; i--) {
            if (up[a][i] != up[b][i]) {
                mxval = max({mxval, mx[a][i], mx[b][i]});
                mnval = min({mnval, mn[a][i], mn[b][i]});
                a = up[a][i];
                b = up[b][i];
            }
        }

        mxval = max({mxval, mx[a][0], mx[b][0]});
        mnval = min({mnval, mn[a][0], mn[b][0]});
        return {up[a][0], mxval, mnval};
    }
};
```

6.10.5 Small To Large

Count the number of occurences of each color in every subtree in $O(nlog(n))$.

sack.cpp

Description: Using small to large technique, copy the big child to parent and iterate small children.

Time: $\mathcal{O}(nlogn)$

d6b8ca, 52 lines

```
vector<vll> g(MAX), vec(MAX);
vector<ll> color(MAX), sz(MAX, 1), cnt(MAX, 0);

// get size of each subtree
void dfsSize(ll u, ll p = -1) {
    for(auto v : g[u]) if (v != p) {
        dfsSize(v, u);
        sz[u] += sz[v];
    }
}

// small to large dfs O(n log(n))
void dfs(ll u, ll p = -1, bool keep=true) {

    // find the biggest child
    ll mx = 0, big = -1;
    for(auto v : g[u]) if (v != p) {
        if (sz[v] > mx) {
            mx = sz[v], big = v;
        }
    }

    // visit all small children
    for(auto v : g[u]) if (v != p and v != big) {
        dfs(v, u, 0);
    }

    // visit big child, get his cnt
    if (big != -1) {
        dfs(big, u, 1);
        swap(vec[u], vec[big]);
    }

    // add itself
    vec[u].pb(u);
    cnt[color[u]] += 1;

    // add small children
    for(auto v : g[u]) if (v != p and v != big) {
        for(auto id : vec[v]) {
            vec[u].pb(id);
            cnt[color[id]] += 1;
        }
    }

    // remove cnt from small children
    if (keep == 0) {
        for(auto id : vec[u]) {
            cnt[ color[id] ] -= 1;
        }
    }
}
```

Mathematics (7)

7.1 Modular Arithmetic

modular.cpp

Description: mint struct for modular arithmetic operations

Time: $\mathcal{O}(1)$ for most operations, $\mathcal{O}(log(n))$ for division and exponentiation

b1a4c8, 98 lines

```
// supports operations between int/ll and mint,
// and it will return a mint object independently of the order
    of operations
template<ll P> struct Z {
    ll val;

    Z(ll a = 0) {
        val = a % P;
        if (val < 0) val += P;
    }

    Z& operator +=(Z rhs) {
        val += rhs.val;
        if (val >= P) val -= P;
        return *this;
    }
    friend Z operator +(Z lhs, Z rhs) { return lhs += rhs; }

    Z& operator -=(Z rhs) {
        val += P - rhs.val;
        if (val >= P) val -= P;
        return *this;
    }
    friend Z operator -(Z lhs, Z rhs) { return lhs -= rhs; }

    Z& operator *=(Z rhs) {
        val = (val * rhs.val) % P;
        return *this;
    }
    friend Z operator *(Z lhs, Z rhs) { return lhs *= rhs; }

    Z operator ^(ll i) const {
        Z ans = 1, aux = val;
        while(i) {
            if (i & 1) ans *= aux;
            aux *= aux;
            i >>= 1;
        }
        return ans;
    }

    Z& operator /=(Z rhs) {
        return *this *= rhs^(P-2);
    }
    friend Z operator /(Z lhs, Z rhs) { return lhs /= rhs; }

    bool operator ==(Z rhs) { return val == rhs.val; }

    bool operator !=(Z rhs) { return val != rhs.val; }

    friend ostream& operator <<(ostream& out, Z a) { return out
        << a.val; }

    friend istream& operator >>(istream& in, Z& a) {
        ll x; in >> x;
        a = Z(x);
        return in;
    }
};
using mint = Z<MOD>;
```

7.1.1 Lucas's Theorem

$$\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$$

For p prime. n_i and m_i are the coefficients of the representations of n and m in base p.

Example:

11 (in base p=3) $= 1 \cdot 3^2 + 0 \cdot 3^1 + 2 \cdot 3^0$

$\implies n_2 = 1, n_1 = 0, n_0 = 2$

7.1.2 Fermat's Little Theorem

Fermat's little theorem states that if p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p:

$$a^p \equiv a \pmod{p}$$

If a is not divisible by p, that is, if a is coprime to p, then Fermat's little theorem is equivalent to:

$$a^{p-1} \equiv 1 \pmod{p}$$

In other words, when doing a double exponentiation. Do:

$$a^{b^c} \pmod{p} \equiv a^{(b^c \pmod{p-1})} \pmod{p}$$

7.2 Combinatorics

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}, \qquad 0 <= m <= n$$

$$0, \qquad \textit{otherwise}$$

7.2.1 Factorial

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800
n	11	12	13	14	15	16	17			
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14			
n	20	25	30	40	50	100	150	171		
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX		

7.2.2 Combinatorial Struct

combinatorics.cpp

Description: basic operations for combinatorics problems under a certain modulo

Time: $\mathcal{O}(n)$ to construct, $\mathcal{O}(1)$ operations

bb55e9, 87 lines

```
// remeber to import mint struct !!
struct Combinatorics {
    vector<mint> fact, ifact;

    Combinatorics(ll n) : fact(n+1), ifact(n+1) { // inclusive
        n
        fact[0] = 1;
        for (ll i=1; i<=n; i++) fact[i] = fact[i-1] * i;

        ifact[n] = 1 / fact[n];
        for (ll i=n; i>0; i--) ifact[i-1] = ifact[i] * i;
    }

    // Combination, "Combinacao"
    // n objects to place in k spaces
    // the order doesn't matter, so we consider the re-
        orderings
    // = n! / (k! * (n-k)!)
    mint C(ll n, ll k) {
        if (k < 0 or n < k) return 0;
        return fact[n] * ifact[k] * ifact[n-k];
    }

    // Permutation, "Permutacao"
    // n objects to place in n spaces
    // = n!
    mint P(ll n) {
        if (n < 0) return 0;
        return fact[n];
    }

    // Permutation with Repetition, "Permutacao com repeticao"
    // Also called: Multinomial coefficients
    // n objects to place in n spaces
    // some objects are equal
    // therefore, we consider the possible re-orderings
    // = n! / (k1! k2! k3!)
    mint PR(ll n, vector<ll> vec) {
        if (n < 0) return 0;
        mint ans = fact[n];
        for(auto val : vec) ans *= ifact[val];
        return ans;
    }

    // Arrangement, "Arranjo Simples"
    // n objects to place in k spaces (k < n)
    // n * (n-1) * ... * (n-k+1)
    // = n! / (n-k)!
    mint A(ll n, ll k) {
        if (n < 0) return 0;
        return fact[n] * ifact[n-k];
    }

    // Stars and Bars, "Pontos e Virgulas"
    // n stars to distribute among
    // k distint groups, that can contain 0, 1 or more stars
    // separated by k-1 bars
    // = (n+k-1)! / ( n! * (k-1)! )
    mint SB(ll n, ll k) {
        if (k == 0) {
            if (n == 0) return 1;
            else return 0;
        }
        return C(n + k - 1, k - 1);
    }

    // a derangement is a permutation of the elements of a set
    // in which no element appears in its original position
    // In other words, a derangement is a permutation that has
        no fixed points.
    // derangement(n) = subfactorial(n) = !n
    // !n = (n-1) * ( !(n-1) + !(n-2) ), for n>= 2
    // !1 = 0, !0 = 1
    vector<mint> subfact;
    void computeSubfactorials(ll n) {
        subfact.assign(n+1, 0);
        subfact[0] = 1;
        subfact[1] = 0;

        for(ll i=2; i<=n; i++) {
            subfact[i] = (i-1) * (subfact[i-1] + subfact[i-2]);
        }
    }

    // remeber to compute subfactorials first !!
    mint derangement(ll n) {
        if (n < 0) return 0;
        return subfact[n];
    }
};
Combinatorics op(MAX); // MAX = inclusive max_value for fact[]
```

7.2.3 Burside Lemma

Let G be a group that acts on a set X. The Burnside Lemma states that the number of distinct orbits is equal to the average number of points fixed by an element of G.

$$T = \frac{1}{|G|} \sum_{g \in G} |\texttt{fix}(g)|$$

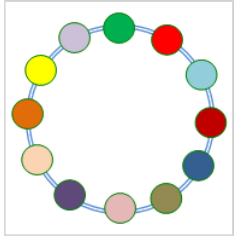
Where a orbit $\texttt{orb}(x)$ is defined as

$$\texttt{orb}(x) = \{y \in X : \exists g \in G \; gx = y\}$$

and $\texttt{fix}(g)$ is the set of elements in X fixed by g

$$\texttt{fix}(g) = \{x \in X : gx = x\}$$

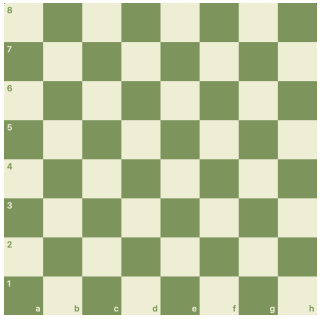
Example1: With k distinct types of beads how many distinct necklaces of size n can be made? Considering that two necklaces are equal if the rotation of one gives the other.



$$\frac{1}{n} \sum_{i=1}^{n} k^{\gcd(i,n)}$$

Example2: Count the number of different n × n grids whose each square is black or white.

Two grids are considered to be different if it is not possible to rotate one of them so that they look the same.



$$G(Rotations) = 0°, 90°, 180°, 270°$$

$$f(rotation) = \begin{cases} 0° : 2^{(n^2)} & \\ 90°/270° : 2^{\frac{n^2}{4}}, & n_{even} \\ \quad 2^{\frac{n^2-1}{4}} \cdot 2, & n_{odd} \\ 180° : 2^{\frac{n^2}{2}}, & n_{even} \\ \quad 2^{\frac{n^2-1}{2}} \cdot 2, & n_{odd} \end{cases}$$

$$ans = \frac{1}{4}(f(0°) + f(90°) + f(180°) + f(270°))$$

7.2.4 Interesting Recursion

$$f(a,b) = f(a-1,b) + f(a,b-1)$$

$$\implies f(a,b) = \frac{(a+b)!}{a!b!} = \binom{a+b}{a}$$

Proof:

$$\begin{aligned} & f(a,b) = \frac{(a+b)!}{a!b!} \\ \implies & f(a-1,b) = \frac{(a-1+b)!}{(a-1)!b!}, f(a,b-1) = \frac{(a+b-1)!}{a!(b-1)!} \\ \implies & f(a-1,b) + f(a,b-1) = \frac{(a-1+b)!}{(a-1)!b!} + \frac{(a+b-1)!}{a!(b-1)!} \\ \implies & f(a,b) = (a+b-1)! \cdot (\frac{1}{(a-1)!(b)!} + \frac{1}{(a)!(b-1)!}) \\ \implies & f(a,b) = (a+b-1)! \cdot (\frac{a+b}{a!b!}) \\ \implies & f(a,b) = \frac{(a+b)!}{a!b!} = \binom{a+b}{a} \end{aligned}$$

7.3 Number Theory

7.3.1 Highly Composite Numbers

<=	Divisors	Number	Factorization
1e3	32	840	$2^3\,3\,5\,7$
1e4	64	7560	$2^3\,3^3\,5\,7$
1e5	128	83160	$2^3\,3^3\,5\,7\,11$
1e6	240	720720	$2^4\,3^2\,5\,7\,11\,13$
1e9	1344	735134400	$2^6\,3^3\,5^2\,7\,11\,13\,17$
1e12	6720	963761198400	$2^6\,3^4\,5^2\,7\,11\,13\,17\,19\,23$
1e14	17280	97821761637600	$2^5\,3^4\,5^2\,7^2\,11\,13\,17\,19\,23\,29$
1e15	26880	866421317361600	$2^6\,3^4\,5^2\,7\,11\,13\,17\,19\,23\,29\,31$
1e18	103680	897612484786617600	$2^8\,3^4\,5^2\,7^2\,11\,13\,17\,19\,23\,29\,31\,37$

7.3.2 Divisibility

"a is divisible by b" or "a is a multiple of b" or "b is a divisor of a" or "b is a factor of a" or "b divides a" or "$(b|a)$"

$$a\%b == 0$$

"a1, a2 are divisible by b"

$$gcd(a1, a2)\%b = 0$$

"a is divisible by b1 and b2"

$$a\%lcm(b1, b2) = 0$$

Euclid

$$a = bq + r$$

Lema 1 - Transitivity

"a is divisible by b and b is divisible by c"

$$(a|b), (b|c) \implies (a|c)$$

Lema 2

$$(a|b), (a|c) => a|(rb + sc)$$

Lema 3

$$d = gcd(a,b) \implies gcd(a/d, b/d) = 1$$

Lema 4

$$d = gcd(a,b) \implies d = ra + sb \implies (d_0|a), (d_0|b) \implies (d_0|d)$$

Lema 5

$$(a|bc), gcd(a,b) = 1 \implies (a|c)$$

Lema 6

$$a = bq + r, 1 <= r < b \implies gcd(a,b) = gcd(b,r)$$

Greatest Common Divisor (GCD)

$$gcd(a) = a$$

$$gcd(a,b,c) = gcd(gcd(a,b),c)$$

$$gcd(a,b) = (a*b)/lcm(a,b)$$

Least Commom Multiple (LCM)

$$lcm(a) = a$$

$$lcm(a,b,c) = lcm(lcm(a,b),c)$$

$$lcm(a,b) = (a*b)/gcd(a,b)$$

Observation: *std-c++17* implements $gcd()$ function, which works correctly for negative numbers as well:

$$gcd(a,b) = gcd(-a,-b) = gcd(-a,b) = gcd(a,-b)$$

7.3.3 Closed Formulas related to divisors of a number

Let n be a number represented by it's prime factors p_i and respective exponents e_i:

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$$

Number of Divisors

$$d(n) = (e_1 + 1) \cdot (e_2 + 1) \cdots (e_k + 1)$$

$$C_i = C_{i-1} \cdot (e_i + 1)$$

Sum of Divisors

$$\sigma(n) = \frac{p_1^{e_1+1} - 1}{p_1 - 1} \cdot \frac{p_2^{e_2+1} - 1}{p_2 - 1} \cdots \frac{p_k^{e_k+1} - 1}{p_k - 1}$$

$$S_i = S_{i-1} \cdot \frac{p_i^{e_i+i} - 1}{p_i - 1}$$

Product of Divisors

$$P_i = (P_{i-1})^{(e_i+1)} \cdot (p_i{}^{(e_i(e_i+1))/2})^{C_{i-1}}$$

where, C_i = "Number of Divisors considering i factors". And remeber to use **Fermat's Little Theorem**

7.3.4 Sieves

These sieves are used to find all primes up to an upper bound N, which is usually 10^7

Eratosthenes

Eratosthenes uses less memory than the linear sieve and is almost as fast

eratosthenes.cpp
Description: Optimized sieve of eratosthenes
Time: $\mathcal{O}(N \log \log N)$
8d74e5, 15 lines

```
// O (N log^2(N) ) -> Teorema de Merten
vector<ll> primes {2, 3};
bitset<MAX> sieve; // {sieve[i] == 1} if i is prime
// MAX can be ~1e7

void eratostenes(ll n){
    sieve.set();
    for(ll i=5, step=2; i<=n; i+=step, step = 6 - step){
        if(sieve[i]){ // i is prime
            primes.pb(i);
            for(ll j= i*i; j<=n; j += 2*i) // sieving all odd
                 multiples of i >= i*i
                sieve[j] = false;
        }
    }
}
```

Linear Sieve

Can check primality with $sp[i] == i$

Uses more memory, because sp is a vector of integers.

linear-sieve.cpp
Description: Linear Sieve that iterates every value once (prime) or twice (composite)
Time: $\mathcal{O}(N)$
251ebd, 17 lines

```
vector<ll> primes, sp(MAX); // MAX = ~1e7
// sp[i] = smallest prime divisor of i
// after running sieve(), sp[i] == i for primes

void sieve(ll n) {
    for (ll i=2; i <= n; i++) {
        if (sp[i] == 0) { // i is prime
            sp[i] = i; // {sp[i] == i} for prime numbers
            primes.pb(i);
        }
        // visit every composite number that has primes[j] as
             the sp
        for (ll j = 0; i * primes[j] <= n; j++) {
            sp[i * primes[j]] = primes[j];
            if (primes[j] == sp[i]) break;
        }
    }
}
```

7.3.5 Factorization

factorization.cpp
Description: Factorization
Time: $\mathcal{O}(\sqrt{n})$ for trial division; $\mathcal{O}(\log 2(n))$ using smallest prime
d1a675, 22 lines

```
vector<ll> factorization(ll n){ // O(sqrt(n))
    vector<ll> factors;
    for(auto p : prime){
        if (p*p > n) break;
        while(n % p == 0){
            factors.pb(p);
            n /= p;
        }
    }
    if (n > 1) factors.pb(n);
    return factors;
}

// import linear sieve
vector<ll> factorization(ll a) { // log2(a)
    vector<ll> factors;
    while(a > 1) {
        factors.pb(sp[a]);
        a /= sp[a];
    }
    return factors;
}
```

7.3.6 Extended Euclid

Solves the $ax + by = gcd(a, b)$ equation.

Inverse Multiplicative

if $gcd(a, b) = 1$:

then:

$$ax + by \equiv 1$$

also, if you apply $\pmod b$ to the equation:

$$ax \pmod b + by \pmod b \equiv 1 \pmod b$$
$$ax \equiv 1 \pmod b$$

In other words, one can find the inverse multiplicative of any number a in modulo b if $gcd(a, b) = 1$

Diofantine Equation

$$ax \equiv c \pmod b$$

if $g = gcd(a, b, c) \neq 1$, divide everything by g.

After this, if $gcd(a, b) = 1$, find a^{-1}, then multiply both sides of the Diofantine equation.

$$x \equiv c * a^{-1} \pmod b$$

After this, one has simply found x

extended-euclid.cpp
Description: Solves the $a * x + b * y = gcd(a, b)$ equation
Time: $\mathcal{O}(\log min(a, b))$
dfd1d4, 24 lines

```
// equation: a*x + b*y = gcd(a, b)
// input: (a, b)
// returns gcd of (a, b)
// also computes &x and &y, which are passed by reference

ll extendedEuclid(ll a, ll b, ll &x, ll &y) {
    x = 1, y = 0;
    ll x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        ll q = a1 / b1;
        tie(x, x1) = pair{x1, x - q * x1};
        tie(y, y1) = pair{y1, y - q * y1};
        tie(a1, b1) = pair{b1, a1 - q * b1};
    }
    return a1;
}

// returns val^(-1) (mod m)
// <=> gcd(val, m) == 1
ll inv(ll val, ll m) {
    ll x, y;
    extendedEuclid(val, m, x, y);
    return ((x % m) + m) % m;;
}
```

7.4 FFT

FFT can be used to turn a polynomial multiplication complexity to $O(N \log N)$.

A **convulution** is easily computed by inverting the second vector and doing the polynomial multiplication normally.

fft-simple.cpp
Description: Computes the product between two polynomials using fft
Time: $\mathcal{O}(N \log N)$
6f735a, 69 lines

```
// #define ld long double
// const ld PI = acos(-1);

struct num {
    ld a {0.0}, b {0.0};
    num() {}
    num(ld na) : a{na} {}
    num(ld na, ld nb) : a{na}, b{nb} {}
    const num operator +(const num &c) const{
        return num(a + c.a, b + c.b);
    }
    const num operator -(const num &c) const{
        return num(a - c.a, b - c.b);
    }
    const num operator *(const num &c) const{
        return num(a*c.a - b*c.b, a*c.b + b*c.a);
    }
    const num operator /(const int &c) const{
        return num(a/c, b/c);
    }
};

void fft(vector<num> &a, bool invert) {
    int n = (int)a.size();
    for(int i=1,j=0;i<n;i++) {
        int bit = n>>1;
        for(; j&bit; bit>>=1)
            j^=bit;
```

```
        j^=bit;
        if(i<j) swap(a[i], a[j]);
    }
    for(int len = 2; len <= n; len <<= 1) {
        ld ang = 2 * PI / len * (invert ? -1 : 1);
        num wlen(cos(ang), sin(ang));
        for(int i=0; i<n; i+=len) {
            num w(1);
            for (int j=0;j<len/2;j++){
                num u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w = w * wlen;
            }
        }
    }
    if(invert) {
        for(num &x: a)
            x = x/n;
    }
}

vector<ll> multiply(vector<int> const& a, vector<int> const& b)
    {
    vector<num> fa(a.begin(), a.end());
    vector<num> fb(b.begin(), b.end());
    int n = 1;
    while(n < int(a.size() + b.size()) )
        n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for(int i=0;i<n;i++)
        fa[i] = fa[i]*fb[i];
    fft(fa, true);
    vector<ll> result(n);
    for(int i=0;i<n;i++)
        result[i] = (ll) round(fa[i].a);
    // while(result.back()==0) result.pop_back();
    return result;
}
```

7.5 Matrix

For faster linear recurrence computation with matrix exponentiation.

$$Base * Operator^{k} = Result$$

matrix.cpp
Description: Matrix structure using vector<vector<T>> as a container.
Time: $\mathcal{O}(n^2)$ for sum, $\mathcal{O}(n^3)$ for product and $\mathcal{O}(n^3 log(k))$ for fexp. fc44ae, 53 lines

```
template<class T> struct Matrix : vector<vector<T>> {
    ll n, m; // rows, columns

    // identity == 0 => Empty matrix constructor
    // identity == 1 => Generates a Identity Matrix (square
        matrix)
    Matrix(ll row, ll col, bool identity = 0) : n(row), m(col)
        {
        (*this).assign(row, vector<T>(col, 0));
        if (identity) {
            assert(n == m);
            for(ll i=0; i<n; i++) (*this)[i][i] = 1;
        }
    }

    // A+B (sizeof(A) == sizeof(B))
    Matrix operator +(const Matrix &b) const { // O(n^2)
        assert(n == b.n and m == b.m);
        Matrix ans(n, m);
        for(ll i=0; i<n; i++){
            for(ll j=0; j<m; j++){
                ans[i][j] = (*this)[i][j] + b[i][j];
            }
        }
        return ans;
    }

    // A*B (A.column == B.row)
    Matrix operator *(const Matrix &b) const { // O(n^3)
        ll n2 = b.n, m2 = b.m;
        assert(m == n2);
        Matrix ans(n, m2);
        for(ll i=0; i<n; i++){
            for(ll j=0; j<m2; j++){
                for(ll k=0; k<n2; k++){
                    ans[i][j] += (*this)[i][k] * b[k][j];
                }
            }
        }
        return ans;
    }

    // A^i (n == m)
    Matrix operator ^(ll i) const{ // O(n^3 log(i))
        assert(n == m);
        Matrix ans(n, n, 1); // identity matrix
        Matrix tmp = *this;
        while(i) {
            if (i & 1) ans = (ans * tmp);
            tmp = (tmp * tmp);
            i >>= 1;
        }
        return ans;
    }
};
```

7.5.1 Minimum Path Length with exactly k edges

Consider a directed weighted graph having n nodes and m edges. Your task is to calculate the minimum path length from node 1 to node n with exactly k edges.

This task is solved the using matrix exponentiation the same way as the problem of **Counting the Number of Paths with exactly k edges**. But there are some modifications in the matrix properties:

The *null* (and default) element is now INF. The identity is composed of 0 in the diagonal. And the product of matrices defined as:

$$AB[i,j] = \min_{k=1}^{n}(AB[i,j], A[i,k] + B[k,j])$$

Finally, the operator matrix contains the value of the minimum weight in each pairwise nodes or INF (if no edges).

7.6 Series' Closed Formulas

7.6.1 Natural Number Summation (PA)

$$1+2+3+4+5+...+n-1+n$$
$$=\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

7.6.2 Natural Number Quadratic Summation

$$1+4+9+16+25+...+(n-1)^2+n^2$$
$$=\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

7.6.3 Triangular Numbers Summation

$$1+3+6+10+15+...+\frac{(n-1)(n)}{2}+\frac{(n)(n+1)}{2}$$
$$=\sum_{i=1}^{n}\frac{i(i+1)}{2} = \frac{1}{2}(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i)$$
$$=\frac{1}{2}(\frac{n(n+1)}{2}+\frac{n(n+1)(2n+1)}{6})$$

7.7 Xor Basis

A *basis* is composed of N **linearly indepedent** vectors, meaning that no combination of other vectors present in the basis can form a reference vector also present in the basis.

The *span* is the set of generatable vectors by a linear combination of the vectors in the basis. The number of vectors present in the span is 2^N, where N is the number of vectors present in the basis. This is because for each vector in the basis, you can include it or not, and each combination will generate an unique vector.

Therefore, each vector in the span has an **unique** combination that forms it.

Given that you have M vectors at your disposal and N of them ($N \leq M$) forms the base, the dimension of the *kernel* is $M - N$, which represents redundant vectors.

For each redundant vector, the numbers of ways to form a vector in the span **doubles**. Consequently, each vector in the span can be generated using $2^(M-N)$ unique linear combinations of the given vectors.

$(red = reduce(val))$ is the reduced form of a vector considering a certain basis.

If the reduced form is 0, that means that this considered vector can already be generated by this basis.

Otherwise, the reduced form of a number is the value that when "xoring" with all the N elements already present in the basis, will generate another N vectors that weren't present in the previous span. Therefore, doubling the number of generatable vectors.

7.7.1 Note:

Instead of integers, bitsets can also be used to for the xor-basis.

For range queries, we can reorder the queries and process them by R. In addition, we create an array of xb[L], that will contain the xor-basis starting from L. This will give a complexity of probably $O(N \log^2 N + Q)$. See code for more details.

7.7.2 Solves:

- Find if a certain vector can be formed by the basis (*if(reduce(val) != 0)*)
- Find how many linear combinations form a certain vector ($ans = 2^{(}dim(kernel))$)
- Find the maximum vector that can be formed ($mx = max(mx, mx\hat{b})$)
- Find the k-th generatable value by the basis (*gaussJordan(), look at the binary representation of* K)

xor-basis.cpp

Description: Xor Basis

Time: $\mathcal{O}(size(base)) = \mathcal{O}(\log mx\_val)$;

538793, 98 lines

```
struct XorBasis {
    vector<ll> B; // basis

    ll reduce(ll vec) {
        for(auto b : B) vec = min(vec, vec^b);
        return vec;
    }

    void add(ll vec) {
        ll val = reduce(vec);
        if (val) B.pb(val);
    }
};

// Extended //
struct XorBasis {
    vector<ll> B;
    ll mx = 0;

    ll reduce(ll vec) {
        if (!vec) return 0;
        for(auto b : B) vec = min(vec, vec^b);
        return vec;
    }

    bool add(ll vec) {
        ll val = reduce(vec);
        if (val) {
            B.pb(val);
            mx = max(mx, mx^val);
            return true;
        }
        return false;
    }

    ll dim() {
        return B.size();
    }

    // Gaussian elimination in O(dim^2)
    // each bit below and above the pivot are zeroed
    // Basis will be ordered from MSB to LSB
    void gaussJordan() {
        sort(B.begin(), B.end(), greater<ll>());
        for(ll i=1; i<(ll)B.size(); i++) {
            for(ll j=0; j<i; j++) {
                B[j] = min(B[j], B[j]^B[i]);
            }
        }
    }
};

// Problem description: (Ivan and Burgers)
// given a static array x[1, N]
// for each query, answer then max xor-sum of any subset in
    subarray [L, R]
// Contrains: 1 <= L <= R <= N, N<= 5e5, Q<= 5e5

// Probably the complexity is O(N log^2(N) + Q)
// Similarly, we can answer other type of queries related to
    xor-basis,
// because we will have it computed (Atcoder: H - Xor Query)
int32_t main(){ sws;
    ll n; cin >> n;

    vector<ll> x(n+1);
    for(ll i=1; i<=n; i++) {
        cin >> x[i];
    }

    vector<vector<pll>> queries(n+1);
    ll q; cin >> q;
    for(ll i=1; i<=q; i++) {
        ll l, r; cin >> l >> r;
        queries[r].pb({l, i});
    }

    vector<XorBasis> xb(n+1); // extended version of XorBasis
    vector<ll> ans(q+1);

    for(ll r=1; r<=n; r++) {

        // O(de bom), maybe log?
        for(ll l=r; l>=1; l--) {
            if (!xb[l].add(x[r])) break;
            // We can break here, because this xor-basis of L
                already contains a basis that doesn't need x[r
                ]
            // Therefore, the xor-basis of L-1, L-2, ..., which
                contains the xor-basis of L, also doesn't
                need x[r]
        }

        // solve all queries ending in r,
        // knowing that all xor-basis are computed up to r.
        for(auto [left, i] : queries[r]) {
            ans[i] = xb[left].mx;
        }
    }

    for(ll i=1; i<=q; i++) {
        cout << ans[i] << endl;
    }
}
```

Strings (8)

8.1 Hashing

Hashing consists in generating a Polynomial for the string, therefore, assigning each distint string to a specific numeric value In practice, there will always be some collisions:

Probability of colision: $= \frac{n^2}{2m}$

n = Comparissons, m = mod size

when using multiple mods, they multiply: m = m1 * m2

hashing.cpp

Description: Create a numerical value for a string by using polynomial hashing

Time: $\mathcal{O}(n)$ to build, $\mathcal{O}(1)$ per query

c3a650, 43 lines

```
// s[0]*P^n + s[1]*P^(n-1) + ... + s[n]*P^0
// 0-idx
struct Hashing {
    ll n, mod;
    string s;
    vector<ll> p, h; // p = P^i, h = accumulated hash sum

    const ll P = 31; // can be 53

    Hashing(string &s_, ll m)
      : n(s_.size()), s(s_), mod(m), p(n), h(n) {

        for(ll i=0; i<n; i++)
            p[i] = (i ? P*p[i-1] : 1) % mod;

        for(ll i=0; i<n; i++)
            h[i] = (s[i] + P*(i ? h[i-1] : 0)) % mod;
    }

    ll query(ll l, ll r) { // [l, r] inclusive (0-idx)
        ll hash = h[r] - (l ? (p[r-l+1]*h[l-1]) % mod : 0);
        return hash < 0 ? hash + mod : hash;
    }
};

// for codeforces:
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());

int32_t main() { sws;
    vector<ll> mods = {
        1000000009,1000000021,1000000033,
        1000000087,1000000093,1000000097,
        1000000103,1000000123,1000000181,
        1000000207,1000000223,1000000241,
        1000000271,1000000289,1000000297
    };

    shuffle(mods.begin(), mods.end(), rng);

    string s; cin >> s;

    Hashing hash(s, mods[0]);
```

```
}
```

8.2 Z-Function

Suppose we are given a string s of length n. The Z-function for this string is an array of length n where the i-th element is equal to the greatest number of characters starting from the position i that coincide with the first characters of s (the prefix of s)

The first element of the Z-function, z[0], is generally not well defined. This implementation assumes it as z[0] = 0. But it can also be interpreted as z[0] = n (all characters coincide).

Can be used to solve the following simples problems:

- Find all ocurrences of a pattern p in another string s. (p + '$' + s) (z[i] == p.size())
- Find all borders. A border of a string is a prefix that is also a suffix of the string but not the whole string. For example, the borders of abcababcab are ab and abcab. (z[8] = 2, z[5] = 5) (z[i] = n-i)
- Find all period lengths of a string. A period of a string is a prefix that can be used to generate the whole string by repeating the prefix. The last repetition may be partial. For example, the periods of *abcabca* are **abc**, **abcabc** and **abcabca**.

  It works because (z[i] + i ¿= n) is the condition when the common characters of z[i] in addition to the elements already passed, exceeds or is equal to the end of the string. For example:

  abaababaab z[8] = 2

  abaababa is the period; the remaining (z[i] characters) are a prefix of the period; and when all these characters are combined, it can form the string (which has n characters).

zfunction.cpp
Description: For each substring starting at position i, compute the maximum match with the original prefix. z[0] = 0
Time: $\mathcal{O}(n)$
14b37c, 12 lines

```
vector<ll> z_function(string &s) { // O(n)
    ll n = (ll) s.length();
    vector<ll> z(n);
    for (ll i=1, l=0, r=0; i<n; i++) {
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);

        while (i + z[i] < n and s[z[i]] == s[i + z[i]]) z[i]++;

        if (r < i + z[i] - 1) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

8.3 KMP

KMP stands for Knuth-Morris-Pratt and computes the prefix function.

You are given a string s of length n. The prefix function for this string is defined as an array π of length n, where $\pi[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $\pi[0] = 0$.

For example, prefix function of string *"abcabcd"* is $[0, 0, 0, 1, 2, 3, 0]$, and prefix function of string *"aabaaab"* is $[0, 1, 0, 1, 2, 2, 3]$.

kmp.cpp
Description: Computes the prefix function
Time: $\mathcal{O}(n)$
3f2929, 13 lines

```
vector<ll> kmp(string &s) { // O(n)
    ll n = (ll) s.length();
    vector<ll> pi(n, 0); // pi[0] = 0
    for (ll i=1; i<n; i++) {
        ll j = pi[i-1];
        while (j > 0 and s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

8.3.1 Patterns in a String

Given a string p (pattern) and a string s, we want to find and display the positions of all occurrences of the string p in the string s.

Solution: Concatenate $p +' \$' + s$, each position where $pi[i] == p.size() \implies$ a match of the pattern in this substring.

8.4 Suffix Array

The suffix array is the array with size n, whose values are the indexes from the longest substring (0) to the smallest substring (n) after ordering it lexicographically. Example:

```
Let the given string be "banana".

0 banana                          5 a
1 anana      Sort the Suffixes    3 ana
2 nana       ---------------->    1 anana
3 ana         alphabetically      0 banana
4 na                              4 na
5 a                               2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

Note that the length of the string i is: (s.size()-sa[i])

suffix-array.cpp
Description: Creates the Suffix Array
Time: $\mathcal{O}(N \log N)$
49608b, 20 lines

```
vector<ll> suffixArray(string s) {
    s += "!";
    ll n = s.size(), N = max(n, 260LL);
    vector<ll> sa(n), ra(n);
    for (ll i = 0; i < n; i++) sa[i] = i, ra[i] = s[i];

    for (ll k = 0; k < n; k ? k *= 2 : k++) {
        vector<ll> nsa(sa), nra(n), cnt(N);

        for (ll i = 0; i < n; i++) nsa[i] = (nsa[i]-k+n)%n, cnt
            [ra[i]]++;
        for (ll i = 1; i < N; i++) cnt[i] += cnt[i-1];
        for (ll i = n-1; i+1; i--) sa[--cnt[ra[nsa[i]]]] = nsa[
            i];

        for (ll i = 1, r = 0; i < n; i++) nra[sa[i]] = r += ra[
            sa[i]] !=
            ra[sa[i-1]] or ra[(sa[i]+k)%n] != ra[(sa[i-1]+k)%n
                ];
        ra = nra;
        if (ra[sa[n-1]] == n-1) break;
    }
    return vector<ll>(sa.begin()+1, sa.end());
}
```

Kasai generates an array of size n (like the suffix array), whose values indicates the lenght of the longest common prefix beetwen (sa[i] and sa[i+1])

kasai.cpp
Description: Creates the Longest Common Prefix array (LCP)
Time: $\mathcal{O}(N \log N)$
913195, 13 lines

```
vector<ll> kasai(string s, vector<ll> sa) {
    ll n = s.size(), k = 0;
    vector<ll> ra(n), lcp(n);
    for (ll i = 0; i < n; i++) ra[sa[i]] = i;

    for (ll i = 0; i < n; i++, k -= !!k) {
        if (ra[i] == n-1) { k = 0; continue; }
        ll j = sa[ra[i]+1];
        while (i+k < n and j+k < n and s[i+k] == s[j+k]) k++;
        lcp[ra[i]] = k;
    }
    return lcp;
}
```

Problems that can be solved:

Numbers of Distinct Substrings:

- $\frac{n(n+1)}{2} - lcp[i]$ (for all i)

Longest Repeated Substring:

- biggest lcp[i]. The position can be found in sa[i]

Find how many distinct substrings there are for each len in [1:n]:

- Use delta encoding and the fact that lcp[i] counts the repeated substring between s.substr(sa[i]) and s.substr(sa[i+1]), which are the substrings corresponding to the commom prefix.

Find the k-th distinct substring:

```
    string s; cin >> s;
    ll n = s.size();

    auto sa = suffix_array(s);
```

```
    auto lcp = kasai(s, sa);

    ll k; cin >> k;

    for(ll i=0; i<n; i++) {
        ll len = n-sa[i];
        if (k <= len) {
            cout << s.substr(sa[i], k) << endl;
            break;
        }
        k += lcp[i] - len;
    }
```

8.5 Manacher

Manacher's Algorithm is used to find all palindromes in a string.

For each substring, centered at i, find the longest palindrome that can be formed.

Works best for odd size string, so we convert all string to odd ones by adding and extra characters between the original ones

Therefore, the value stored in the vector cnt is actually palindrome-len + 1.

manacher.cpp
Description: Covert String to odd length to use manacher, which computes all the maximum lengths of all palindromes in the given string
Time: $\mathcal{O}(2n)$
0c2a2b, 46 lines

```
struct Manacher {
    string s, t;
    vector<ll> cnt;

    // t is the transformed string of s, with odd size
    Manacher(string &s_) : s(s_) {
        t = "#";
        for(auto c : s) {
            t += c, t += "#";
        }
        count();
    }

    // perform manacher on the odd string
    // cnt will give all the palindromes centered in i
    // for the odd string t
    void count() {
        ll n = t.size();
        string aux = "$" + t + "^";
        vector<ll> p(n + 2);
        ll l = 1, r = 1;
        for(ll i = 1; i <= n; i++) {
            p[i] = max(0LL, min(r - i, p[l + (r - i)]));
            while(aux[i - p[i]] == aux[i + p[i]]) {
                p[i]++;
            }
            if(i + p[i] > r) {
                l = i - p[i], r = i + p[i];
            }
        }
        cnt = vector<ll>(p.begin() + 1, p.end() - 1);
    }

    // compute a longest palindrome present in s
    string getLongest() {
        ll len = 0, pos = 0;
        for(ll i=0; i<(ll)t.size(); i++) {
            ll sz = cnt[i]-1;
            if (sz > len) {
                len = sz;
                pos = i;
            }
        }
        return s.substr(pos/2 - len/2, len);
    }
};
```

8.6 Booth

An efficient algorithm which uses a modified version of KMP to compute the least amount of rotation needed to reach the **lexicographically minimal string rotation**.

A rotation of a string can be generated by moving characters one after another from beginning to end. For example, the rotations of *acab* are *acab*, *caba*, *abac*, and *baca*.

booth.cpp
Description: Use a modified version of KMP to find the lexicographically minimal string rotation
Time: $\mathcal{O}(n)$
64184b, 30 lines

```
// Booth Algorithm
ll least_rotation(string &s) { // O(n)
    ll n = s.length();
    vector<ll> f(2*n, -1);
    ll k = 0;
    for(ll j=1; j<2*n; j++) {
        ll i = f[j-k-1];
        while(i != -1 and s[j % n] != s[(k+i+1) % n] ) {
            if (s[j % n] < s[(k+i+1) % n])
                k = j - i - 1;
            i = f[i];
        }
        if (i == -1 and s[j % n] != s[(k+i+1) % n] ) {
            if (s[j % n] < s[(k+i+1) % n])
                k = j;
            f[j - k] = -1;
        }
        else
            f[j - k] = i + 1;
    }
    return k;
}

int32_t main(){ sws;
    string s; cin >> s;
    ll n = s.length();
    ll ans_idx = least_rotation(s);
    string tmp = s + s;
    cout << tmp.substr(ans_idx, n) << endl;
}
```

8.7 Aho-Corasick

The complexity for following the suffix path upwards is $O(\sqrt{(n)})$ when the summation of patterns is limited in n.

aho.cpp
Description: É aho porra, acha todos os match de stringzinhas em stringzonas
Time: $\mathcal{O}(n)$ add(), $\mathcal{O}(nA)$ init()
1aca17, 42 lines

```
const int A = 26;
int to[N][A];
int ne = 2, fail[N];
vector<int> term[N];
vector<int> g[N];

void add(string str, int id){
    int p = 1;
    for(auto c: str){
        int ch = c - 'a'; // !
        if(!to[p][ch]) to[p][ch] = ne++;
        p = to[p][ch];
    }
    term[p].push_back(id);
}

int compress(int x){
    if(term[x].size() > 0 or x == 1){
        return x;
    }
    return fail[x] = compress(fail[x]);
}

void init(){
    for(int i = 0; i < ne; i++) fail[i] = 1;
    queue<int> q; q.push(1);
    int u, v;
    while(!q.empty()){
        u = q.front(); q.pop();
        for(int i = 0; i < A; i++){
            if(to[u][i]){
                v = to[u][i]; q.push(v);
                if(u != 1){
                    fail[v] = to[ fail[u] ][i];
                }
            }
            else if(u != 1) to[u][i] = to[ fail[u] ][i];
            else to[u][i] = 1;
        }
        g[fail[u]].push_back(u);
    }
}
```

aho2.cpp
Description: É aho tbm porra
Time: $\mathcal{O}(n)$ add(), $\mathcal{O}(nAlpha)$ build()
abc5f0, 65 lines

```
// obs: O(alphabet) is considered constant
const ll alphabet = 27; // index #26 = char('{') (separator)

struct Aho {
    struct State {
        // suffix link is the longest proper suffix
        // exit link is the next marked terminal node in the
            suffix link path
        ll link = 0, exit = 0, depth = 0;
        vector<ll> vec;
        bool term = 0; // isTerminal
        array<ll, alphabet> down = {}; // 0 => non existent
            edge

        ll& operator [](const char &c) {
            return down[c-'a'];
        }
    };

    ll n = 2; // number of states
    vector<State> t; // tree
```

```
    // root = node 1, root.link = 1
    Aho() : t(2) {}

    void add(string &s, ll val) {
        ll u = 1, h = 1;
        for(auto c : s) {
            if (!t[u][c]) {
                t[u][c] = n++;
                t.pb(State{});
            }
            u = t[u][c];
            t[u].depth = h++;
        }
        t[u].term = true;
        t[u].vec.pb(val);
    }

    void build() { // O(n * alphabet)
        for(ll i=1; i<n; i++) {
            t[i].link = 1;
        }
        queue<ll> q; q.push(1);
        while(q.size()) { // bfs
            auto u = q.front(); q.pop();
            for(ll i=0; i<alphabet; i++) {
                char c = char('a' + i);
                if (t[u][c]) {
                    auto v = t[u][c]; q.push(v);
                    if (u != 1) {
                        auto &link = t[v].link;
                        link = t[t[u].link][c];
                        t[v].exit = (t[link].term) ? link : t[
                            link].exit;
                    }
                }
                // if there are no direct edges, use suffix
                    link
                else if (u != 1) {
                    t[u][c] = t[t[u].link][c];
                }
                else { // root
                    t[u][c] = 1;
                }
            }
        }
    }
};
```

s

8.8 Suffix Automaton

8.8.1 Concepts:

The goat!!!

- All substrings of the string s can be decomposed into equivalence classes according to their end positions *endpos*.
- The *endpos* is a subset of positions (0-idx) of s that contains exactly all the end postitions (of the last character) in which there is an occurence of this class of substrings (all of them at once).
- Each unique substring will be represented by exactly one vertex and each vertex (except root) will represent one or more substrings, which are all $endpos - equivalent$.
- In the implementation, due to contrains, there is a variable called *endpos*, which has the cardinality of the set instead of the set itself. and the characters of this substring can be obtained transversing from the root to this node adding all characters from the edges.
- All paths from the root creates an unique substring, and the terminal node reached by this path transversal represents this substring.
- Therefore, all substrings represented in a node are actually paths in the automaton starting from the root and ending at this node.
- A vertex can then be represented by the longest substring with length *len*.
- The suffix link of a node u points to the node that cointains a bigger subset $endpos(link(u))$, that contains all position from $endpos(u)$ ($endpos(u) \subset endpos(link(u))$). Naturally, the root has the set of all positions.
- The substrings represented by a node are suffixes of each other (each one smaller by one), whose length $\in [minlen, len]$,
- If we start from an arbitrary state u and follow the suffix links, eventually we will reach the root. In this case we obtain a sequence of disjoint intervals $[minlen(u_i); len(u_i)]$, which in union forms the continuous interval $[0; len(v_0)]$.
- The *minlen* can be stored implicitly, because $minlen(u) = len(link(u)) + 1$.
- The *fpos* attribute represents the minimal element in the *endpos* set. In other words, the first endpos.
- Considering only the edges in *down*, the automaton is a **DAG**. Considering only the edges in *link*, the automaton is a **tree**.
- Some nodes are called marked as **terminal states**, which represent the suffixes of the main string s. The terminal states are achieved starting from the node of s and following the links until the root. The node containing s is a terminal state and the root isn't.

The number of vertices that are created is upper bounded by $O(2n)$ and the number of edges is bounded by $O(3n)$.

8.8.2 Implementation:

The implementation can be changed to use a a map instead of a fixed vector for adjacent edges. This will increase the time complexity to $O(n \log k + constantofmap)$ and the memory to can be sparse.

suffix-automaton.cpp

Description: Suffix automaton, each node represents a set of end-pos equivalent substrings. Solves A LOT of tasks!
Time: $\mathcal{O}(n)$ to create all nodes, $\mathcal{O}(nlogn)$ to compute endpos size

1423c, 247 lines

```
// obs: O(alphabet) is considered constant
const ll alphabet = 27; // index #26 = char('{') (separator)

struct Automaton {
    struct State {
        ll link = 1, len = 0;
        array<ll, alphabet> down = {}; // 0 => non existent
            edge
        ll endpos = 0, fpos = -1;

        ll& operator [](const char &c) {
            return down[c-'a'];
        }
    };

    ll n = 2; // number of states
    vector<State> ton; // short for automaton :D
    string s;

    Automaton(string ss) : s(ss) {
        // root = 1, root.link = 0 (0 is a dummy node)
        ton.assign(2, {0});
        for(auto c : s) add(c);

        // build(); // remove if O(nlogn) is too much (s.size()
            ~ 2e6)
    }

    vector<pair<ll, ll>> order; // nodes ordered by len (
        decreasing)
    void build() { // compute endpos O(n log(n))
        for(ll i=1; i<n; i++) {
            order.pb({ton[i].len, i});
        }
        sort(order.rbegin(), order.rend());
        for(auto [len, i] : order) {
            ton[ ton[i].link ].endpos += ton[i].endpos;
        }
    }

    ll minlen(ll u) {
        return 1 + ton[ ton[u].link ].len;
    }

    ll last = 1;
    void add(char c) {
        ll u = n++;
        ll p = last;
        last = u;

        State node; // state[u]
        node.len = ton[p].len + 1;
        node.endpos = 1;
        node.fpos = node.len - 1;
        ton.pb(node);

        for (;p and !ton[p][c]; p = ton[p].link)
```

```
            ton[p][c] = u;

        if (p == 0) return;

        ll q = ton[p][c];
        if (ton[p].len + 1 == ton[q].len) {
            ton[u].link = q;
            return;
        }

    ll clone = n++;
        State node2 = ton[q]; // state[clone]
        node2.endpos = 0;
        node2.len = ton[p].len + 1;
        ton.pb(node2);

    ton[u].link = ton[q].link = clone;

    for (; ton[p][c] == q; p = ton[p].link)
            ton[p][c] = clone;
}

    // ------ //
    // Tasks //
    // ------ //

    // s1. Number of distinct substrings
    // separated in a vector by their lengths
    // knowing that a state[u] cover all the substrings (
        suffixes)
    // of size [minlen, len] represented by this state
    // Obs: for non-distinct substrings, the histogram is
        simply n, n-1, ... , 2, 1

    vector<ll> histogram() { // O(n)
        ll sz = s.size();
        vector<ll> ans(sz+1, 0);

        for(ll i=2; i<n; i++) {
            ll mnlen = minlen(i);
            ll len = ton[i].len;

            ans[mnlen] += 1;
            if (len + 1 <= sz)
                ans[len + 1] -= 1;
        }

        // delta encoding
        for(ll len=1; len<=sz; len++) {
            ans[len] += ans[len-1];
        }
        // ans[0] = 0, because the empty string is not
            considered as a substring
        return ans;
    }

    // s2. Find the lexicographically k-th substring (one can
        consider only the distincts or not)
    // The k-th substring corresponds to the lexicographically
        smallest one,
    // which is also the k-th path in the suffix automaton

    // Additionally, by creating the automaton on the
        duplicated string (S+S),
    // the k-th substring with k = s.size(), will give us the
        Smallest cyclic shift (Minimal Rotation)
    // For huge strings, remeber to not build() endpos which is
        O(n logn)

    // ps: number of substrings below node (including node)
    // ps[0] -> include repeated substring, ps[1] -> consider
        only distinct
    vector<ll> ps[2];

    void buildPS() { // O(n)
        assert(!order.empty()); // assert if build() was called

        ps[0].assign(n, 0), ps[1].assign(n, 0);

        for(ll k : {0, 1}) {
            for(auto [len, u] : order) {
                if (u != 1) {
                    ps[k][u] = (k ? 1 : ton[u].endpos);
                }

                for(auto v : ton[u].down) if (v) {
                    ps[k][u] += ps[k][v];
                }
            }
        }
    }

    string substring(ll k, bool distinct = true) { // O(V+E) =
        O(2sz+ 3sz) = O(5sz), sz = s.size()
        assert(!ps[0].empty()); // assert if buildPS() was
            called

        string ans = ""; // {k = 0} will return the empty
            string ""

        function<void (ll)> dfs = [&](ll u) {
            if (k <= 0) return;
            for(ll inc = 0; inc<alphabet; inc++) {
                char c = char('a' + inc);

                ll v = ton[u][c];
                if (!v) continue;

                ll sum = ps[distinct][v];

                if (k <= sum) {
                    ans += c;
                    k -= (distinct ? 1 : ton[v].endpos);
                    dfs(v);
                    if (k <= 0) return;
                }
                else k -= sum; // optimization
            }
        };

        dfs(1);
        return ans;
    }

    // --------- //
    // Patterns //
    // --------- //

    // p1. Check for occurrence of a pattern P
    // by returning the length of the longest prefix of P in S
    // A match occurs when len(prefix_pattern) == len(pattern)

    ll prefixPattern(string &p) { // O( p.size() )
        ll ans = 0, cur = 1;
        for(auto c : p) {
            if (ton[cur][c]) {
                cur = ton[cur][c];
                ans += 1;
            }
            else break;
        }
        return ans;
    }

    // p2. Count the numbers of occurrences of a pattern P

    ll countPattern(string &p) { // O( p.size() )
        assert(!order.empty()); // check if build() was called
        ll u = 1;
        for(auto c : p) {
            if (ton[u][c]) {
                u = ton[u][c];
            }
            else return 0; // no match
        }
        return ton[u].endpos;
    }

    // p3. Find the first position in which occurred the
        pattern (0-idx)

    ll firstPattern(string &p) { // O( p.size() )
        ll u = 1;
        for(auto c : p) {
            if (ton[u][c]) {
                u = ton[u][c];
            }
            else return -1; // no match
        }
        ll sz = p.size();
        return ton[u].fpos - sz + 1;
    }

    // p4. Longest Common Substring of P and S
    // In addition to returning the lcs,
    // it returns an dp array with the lcs size for each end
        position i

    string lcs(string &p, vector<ll> &dp) { // O( p.size() )
        dp.assign(p.size(), 0);

        ll u = 1, match = 0, best = 0, pos = 0;

        for(ll i=0; i<(ll)p.size(); i++) {
            auto c = p[i];

            while(u > 1 and !ton[u][c]) { // no edge -> follow
                link
                u = ton[u].link;
                match = ton[u].len;
            }

            if (ton[u][c]) {
                u = ton[u][c];
                match++;
            }

            dp[i] = match;
            if (match > best) {
                best = match;
                pos = i;
            }
        }

        return p.substr(pos - best + 1, best);
    }
};
```

Miscellaneous (9)

9.1 Ternary Search

ternary-search.cpp
Description: Computes the min/max for a function that is monotonically increasing then decreasing or decreasing then increasing.
Time: $\mathcal{O}(N \log N_3)$
c3a5d7, 48 lines

```
/*
Float and Min Version: Requires EPS (precision usually defined
     in the question text)
*/

ld f(ld d){
    // function here
}

// for min value
ld ternary_search(ld l, ld r){
    while(r - l > EPS){
        // divide into 3 equal parts and eliminate one side
        ld m1 = l + (r - l) / 3;
        ld m2 = r - (r - l) / 3;
        if (f(m1) < f(m2)){
            r = m2;
        }
        else {
            l = m1;
        }
    }
    return f(l); // check here for min/max
}

/*
Integer and Max Version:
*/

ll f(ll idx) {
    // function here
}

// for max value, using integer idx
ll ternary_search(ll l, ll r) {
    while(l <= r) {
        // divide into 3 equal parts and eliminate one side
        ll m1 = l + (r-l)/3;
        ll m2 = r - (r-l)/3;
        if(f(m1) < f(m2)) {
            l = m1+1;
        }
        else {
            r = m2-1;
        }
    }
    return f(l); // check here for min/max
}
```

9.2 Random Generator

random.cpp
Description: Good randomizer to generate int in a range or shuffle vectors
Time: $\mathcal{O}(1)$ for randint, $\mathcal{O}(nlog(n))$ for shuffle
55c5b9, 13 lines

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());

// or for 64 bits
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().
    count());

// to shuffle a vector
vector<int> vec;
shuffle(vec.begin(), vec.end(), rng);

// to limit the number to the range [l, r]
int randint(int l, int r) {
    return (rng() % (r-l+1)) + l;
}
```

9.3 Read an Fraction Input

```
char c;
ll num, den;
cin >> num >> c >> den;
```

9.4 Getline

When consuming whitespace-delimited input (e.g. int n; *cin >> n;*) any whitespace that follows, including a newline character, will be left on the input stream.

Then when switching to line-oriented input, the first line retrieved with getline will be just that whitespace. In the likely case that this is unwanted behaviour.

In other words, *getline*() will consume the whole line, *cin >>* will consume up to the last whitespace (not included)

getline.cpp
Description: Getline code example
Time: $\mathcal{O}(1)$
d9a714, 7 lines

```
int32_t main() {
    // ws is input manipulator to retrieve the whitespace
        character
    ll n; cin >> n >> ws;
    string line;
    // the second line is therefore stored in the object "line
        ". The default delimiter \n is not stored.
    getline(cin, line);
}
```

9.5 Merge Sort

merge-sort.cpp
Description: Sort a vector V and compute the number of inversions needed to sort it (the same number of swaps in a bubble sort)
Time: $\mathcal{O}(N)$ for merge, $\mathcal{O}(N \log(N))$ for merge-sort
19b92b, 36 lines

```
ll merge(vector<ll> &v, ll l, ll r) {
    ll i = l, mid = (l+r)/2, j = mid+1, swaps = 0;
    vector<ll> ans;

    while(i <= mid or j <= r) {

        if(j > r or (v[i] <= v[j] and i <= mid)) {
            ans.pb(v[i]);
            i += 1;
        }

        else if(i > mid or (v[j] < v[i] and j <= r)){
            ans.pb(v[j]);
            j += 1;
            swaps += (mid-i)+1;
            // mid-i+1 = elements remaining in the left
                subarray
            // (same number of elements that will be swaped to
                the right)
        }
    }

    for(ll k=l; k<=r; k++) v[k] = ans[k-l];

    return swaps;
}

// sort [l, r] (inclusive), 0-idx
ll merge_sort(vector<ll> &v, ll l, ll r){
    if(l == r) return 0;

    ll mid = (l+r)/2, swaps = 0;
    swaps += merge_sort(v, l, mid);
    swaps += merge_sort(v, mid+1, r);
    swaps += merge(v, l, r);

    return swaps;
}
```

9.6 Rounding Half to Even

Rounding Half to Even rule states that if fractional part of x is 0.5 (exactly midpoint), then the correct rounded value of x is the even *integer* nearest to x.

half-to-even.cpp
Description: How to round a long double with the rule of "rounding half to even".
Time: $\mathcal{O}(1)$
d45637, 4 lines

```
// 1e6 -> round considering 6 decimal places
ans *= 1e6;
ans = nearbyintl(ans);
ans /= 1e6;
```

9.7 Count Bits in a range

count-bits.cpp
Description: count the numbers with bit b set in range [0, n] and [l, r]
Time: $\mathcal{O}(1)$
5fbbb8, 15 lines

```
// count of numbers with bit b set in range [0, n]
ll sum(ll n, ll b) {
    ll t = 1LL << (b+1);
    ll ans = ((n+1)/t) * (t/2);
    ll last = n + 1 - ((n+1)/t) * t;
    ll ones = max(0LL, last - t/2);
    return ans + ones;
}

// count of numbers with bit b set in range [l, r]
ll range(ll l, ll r, ll b) {
    ll ans = sum(r, b);
    if (l) ans -= sum(l-1, b);
    return ans;
}
```

9.8 Count Digits in a range

count-digits.cpp
Description: count the number of digits d used in range [1, n]
Time: $\mathcal{O}(18)$
2984e7, 13 lines

```
// count the number of digits d used in range [1, n]
ll count_digits(ll n, ll d) {
    ll cnt = 0, t = 1;
    for(ll i=0; i<18; i++, t *= 10) {
        ll sz = n + 1 - t * ( (d == 0) ? 10 : d );
        if (sz <= 0) break;
```

```
        ll intervals = sz / (t * 10);
        ll use = intervals * t;
        ll last = sz - intervals * t * 10;
        cnt += use + max(0LL, min(last, t));
    }
    return cnt;
}
```

9.9 Couting Bishops

bishops.cpp

Description: Count the number of ways k bishops can be placed on an n × n chessboard so that no two bishops attack each other.

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

da94bb, 37 lines

```
// Solution Notes:
// Give a odd index to black diagonals, and even index to white
      diagonals
// by symmetry, index the '/' diagonals.
// dp[i][j] -> the number of ways to place j bishops
     considering the first i diagonals of the same color
// squares(d) -> how many squares are in this diagonal of index
      i
using mint = Z<MOD>;

ll squares(ll d) {
    if (d & 1) return d/4 * 2 + 1;
    else return (d-1)/4 * 2 + 2;
}

int32_t main(){ sws;
    ll n, k; cin >> n >> k;

    vector<vector<bool>> vis(2*n, vector<bool>(2*n, false));
    vector<vector<mint>> tab(2*n, vector<mint>(2*n, 0));

    function<mint (ll, ll)> dp = [&](ll i, ll j) -> mint {
        if (j >= 2*n) return 0;
        if (j == 0) return 1;
        if (i <= 0) return 0;

        if (vis[i][j]) return tab[i][j];
        vis[i][j] = 1;

        mint ans = dp(i-2, j) + dp(i-2, j-1) * (squares(i) - (j
             -1));
        return tab[i][j] = ans;
    };

    mint ans = 0;
    for(ll j=0; j<=k; j++) {
        ans += dp(2*n-1, j) * dp(2*n-2, k-j);
    }

    cout << ans << endl;
}
```

Techniques (A)

techniques.txt 171 lines

```
Fix answer
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
  Square Root
  Harmonic Series
Greedy algorithm
  Scheduling
  Max contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Transform edges into vertices, duplicating the nodes of the
      graph
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  Bellman-Ford
  Konig's theorem and vertex cover
  Independent Set
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programming
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Slope trick
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
XorBasis
FFT
  Polinomial Product
  Convolution
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Z-String
  Tries
  Rolling polynomial hashes
  Suffix array
  Suffix tree
  Suffix Automata
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```