



Universidade de Brasília

# Tomate Cerveja

Wallace Wu, Pedro Gallo, Henrique Ramos

2024-04-28

- 1 Contest
- 2 Data structures
- 3 Dynamic Programming
- 4 Game theory
- 5 Geometry
- 6 Graph
- 7 Mathematics
- 8 Number theory
- 9 Strings
- 10 Miscellaneous

# Contest (1)

template.cpp33 lines

```
// #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)

#define endl '\n'
#define ll long long
#define ld long double
#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))

template<class A>void db(A a){for(auto b:a){cout<<b<<" ";}cout<<endl;}
template<class A>void debug(A a){for(auto b:a)db(b);}
template<class... A>void dbg(A const&...a){((cout<<"{"<<a<<" "
),...);cout<<endl;}

const ll MAX = 1e6+10;
const ll MOD = 1e9 + 7;
const ll INF = INT64_MAX;

#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;
}
```

1 .bashrc1 lines

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

6 hash.sh3 lines

9 # Hashes a file, ignoring all whitespace and comments. Use for # verifying that code was correctly typed. CTRL+D to send EOF

10 cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum | cut -c -6

11

19 troubleshoot.txt52 lines

19 Pre-submit:

21 Write a few simple test cases if sample is not enough.

21 Are time limits close? If so, generate max cases.

21 Is the memory usage fine?

21 Could anything overflow?

21 Make sure to submit the right file.

25

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` !

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:  $O(nm)$ 461370, 54 lines

// 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?

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

// add an end point with heigth 0 to compute the last rectangles

x.pb(0);

11 area = 0;

11 n = x.size();

stack<pll, vector<pll>> st; // {maxLeft, height for this rectangle}

for(ll i=0; i<n; i++) {

11 h = x[i];

11 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;

11 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;

}

}

11 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:**  $O(n)$

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?*

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

    // add an end point with heighh 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:**  $O(\log(N))$  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:**  $O(N\log(N))$ , or  $O(N\log^2(N))$  if iterating the map directly.

6651d7, 25 lines

**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:**  $O(\log(N))$  per insertion, slow constant

```
// 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;
            // guarantees 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:**  $O(\alpha)$

c70430, 62 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;
    }
};

// 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:**  $O(n \log^2(n))$

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

efeb7e, 40 lines

// MAX = maximum number of nodes that can be created

```
struct Trie{
    ll trie[MAX][26];
    bool isWordEnd[MAX];
    ll nxt = 1, wordsCnt = 0;

    void add(string s){ // O(Depth)
        ll node = 0;
        for(auto c: s) {
            if(trie[node][c-'a'] == 0) { // create new node
                trie[node][c-'a'] = nxt++;
            }
            node = trie[node][c-'a'];
        }
        if(!isWordEnd[node]){
            isWordEnd[node] = true;
            wordsCnt++;
        }
    }

    bool find(string s, bool remove=false){ // O(Depth)
        ll node = 0;
        for(auto c: s) {
            if(trie[node][c-'a'] == 0) {
                return false;
            }
            else {
                node = trie[node][c-'a'];
            }
        }
        if(remove and isWordEnd[node]){
            isWordEnd[node] = false;
            wordsCnt--;
        }
        return isWordEnd[node];
    }
};
```

## 2.7 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 offline approach, ordering the queries by L for example.

### 2.7.1 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.

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

156cd2, 71 lines

// [0, n] segtree for range sum query, point increase  
// 0 or 1-idx  
ll L=0, R;

```
struct Segtree {

    struct Node {
        // null element:
        ll ps = 0;
    };

    vector<Node> tree;
    vector<ll> v;

    Segtree(ll n) {
        R = n;
        v.assign(n+1, 0);
        tree.assign(4*(n+1), Node{});
    }

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

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

    void increase(ll idx=1, ll val=0, ll l=L, ll r=R, ll i=1 )
    {
        if (l == r) {
            // increase operation:
            tree[i].ps += val;
        }
        else {
            ll mid = (l+r)/2;
            if (idx <= mid) increase(idx, val, l, mid, 2*i);
            else increase(idx, val, mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }

    Node query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1) {
        // left/right are the range limits for the query
        // l / r are the internal variables of the tree
        if (right < l or r < left){
            // null element:
            return Node{};
        }
        else if (left <= l and r <= right) return tree[i];
        else{
            ll mid = (l+r)/2;
            return merge(
                query(left, right, l, mid, 2*i),
                query(left, right, mid+1, r, 2*i+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.

**Time:**  $\mathcal{O}(N \log N)$  to build,  $\mathcal{O}(\log N)$  to update or query

fb9b46, 71 lines

```
// [0, n] segtree for point update, range min/max query
// 0 or 1-idx
ll L=0, R;
struct Segtree {

    struct Node {
        // null element:
        ll mn = INF, mx = -INF;
    };

    vector<Node> tree;
    vector<ll> v;

    Segtree(ll n) {
        R = n;
        v.assign(n+1, 0);
        tree.assign(4*(n+1), Node{});
    }

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

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

    void update(ll idx=1, ll val=0, ll l=L, ll r=R, ll i=1 ) {
        if (l == r) {
            // increase operation:
            tree[i].mn = tree[i].mx = val;
        }
        else {
            ll mid = (l+r)/2;
            if (idx <= mid) update(idx, val, l, mid, 2*i);
            else update(idx, val, mid+1, r, 2*i+1);
            tree[i] = merge(tree[2*i], tree[2*i+1]);
        }
    }

    Node query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1) {
        if (right < l or r < left){
            // null element:
            return Node{};
        }
        else if (left <= l and r <= right) return tree[i];
        else{
            ll mid = (l+r)/2;
```

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

```

return merge(
    query(left, right, l, mid, 2*i),
    query(left, right, mid+1, r, 2*i+1)
);
}
}
};

```

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.

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

// [0, n] segtree for point query stored value, range increase  
// 0 or 1-idx

ll L=0, R;  
struct Segtree {

```

struct Node {
    // null element:
    ll ps = 0;
};

```

```

vector<Node> tree;
vector<ll> v;

```

```

Segtree(ll n) {
    R = n;
    v.assign(n+1, 0);
    tree.assign(4*(n+1), Node{});
}

```

```

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

```

```

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

```

```

void increase(ll left, ll right, ll val=0, ll l=L, ll r=R,
    ll i=1) {
    if (right < l or r < left) {
        return;
    }
    else if (left <= l and r <= right) {
        // increase operation
    }
}

```

## seg-inverted seg-pa treap

```

tree[i].ps += val;
}
else {
    ll mid = (l+r)/2;
    increase(left, right, val, l, mid, 2*i);
    increase(left, right, val, mid+1, r, 2*i+1);
}
}

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

```

## 2.7.3 PA Segtree

### seg-pa.cpp

**Description:** Seg with PA (Progressao Aritmetica / Arithmetic Progression). When initializing the segment tree, remember to choose the range limits (L, R) and call build().

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

// [0, n] segtree for range sum query, point increase

ll L=0, R;

struct SegtreePA {

```

struct Node {
    // null element:
    ll ps = 0;
};

```

```

vector<Node> tree;
vector<ll> v;
vector<pll> lazy; // {x, y} of {x*i + y}
// x = razao da PA, y = constante

```

```

SegtreePA(ll n) {
    R = n;
    v.assign(n+1, 0);
    tree.assign(4*(n+1), Node{});
    lazy.assign(4*(n+1), pll());
}

```

```

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

```

```

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

```

```

void build(ll l=L, ll r=R, ll i=1) {
    if (l == r) {
        tree[i] = Node {
            // leaf element:
            v[l]
        };
    }
}

```

```

}
else {
    ll mid = (l+r)/2;
    build(l, mid, 2*i);
    build(mid+1, r, 2*i+1);
    tree[i] = merge(tree[2*i], tree[2*i+1]);
}
lazy[i] = {0, 0};
}

void prop(ll l=L, ll r=R, ll i=1) {
    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 };
    tree[i] = merge(tree[i], val);

    if (l != r) {
        ll mid = (l+r)/2;
        lazy[2*i] = sum(lazy[2*i], lazy[i]);
        lazy[2*i+1] = sum(lazy[2*i+1], {x, y + x*(mid-l+1)});
    }

    lazy[i] = {0, 0};
}

// left/right are the range limits for the query
// l / r are the internal variables of the tree
void increase(ll left, ll right, ll x, ll y, ll l=L, ll r=R,
    ll i=1) {
    prop(l, r, i);
    if (right < l or r < left) return;
    else if (left <= l and r <= right) {
        lazy[i] = {x, y};
        prop(l, r, i);
    }
    else {
        ll mid = (l+r)/2;
        increase(left, right, x, y, l, mid, 2*i);
        ll ny = y + max(x*(mid-max(left, l) + 1), 0LL);
        increase(left, right, x, ny, mid+1, r, 2*i+1);
        tree[i] = merge(tree[2*i], tree[2*i+1]);
    }
}

Node query(ll left=L, ll right=R, ll l=L, ll r=R, ll i=1) {
    prop(l, r, i);
    if (right < l or r < left) {
        // null element:
        return Node{};
    }
    else if (left <= l and r <= right) return tree[i];
    else {
        ll mid = (l+r)/2;
        return merge(
            query(left, right, l, mid, 2*i),
            query(left, right, mid+1, r, 2*i+1)
        );
    }
}
};

```

## 2.8 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 || !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:

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



where  $C(k, j)$  is a cost function and  $dp(i, j) = 0$  when  $j \leq 0$  (using 1-idx).

Say  $0 \leq i < m$  and  $1 \leq j \leq n$ , and evaluating  $C$  takes  $O(1)$  time. Then the straightforward evaluation of the above recurrence is  $O(mn^2)$ . There are  $m \times n$  states, and  $n$  transitions for each state.

Let  $opt(i, j)$  be the value of  $k$  that minimizes the above expression. Assuming that the cost function satisfies the quadrangle inequality, we can show that  $opt(i, j - 1) \leq opt(i, j)$  for all  $i, j$ . This is known as the monotonicity condition. Then, we can apply divide and conquer DP. The optimal "splitting point" for a fixed  $i$  increases as  $j$  increases.

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

To minimize the runtime, we apply the idea behind divide and conquer. First, compute  $opt(i, n/2)$ . Then, compute  $opt(i, n/4)$ , knowing that it is less than or equal to  $opt(i, n/2)$ ; and  $opt(i, 3n/4)$  knowing that it is greater than or equal to  $opt(i, n/2)$ .

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

divide-conquer-dp.cpp  
**Description:** Optimize an  $O(mn^2)$  dp to  $O(mn \log(n))$  using divide and conquer. cost function must have quadrangle inequality ("wider is worse")  
**Time:**  $O(mn \log(n))$

```
// m partitions, n elements
11 divideConquerDP(11 m, vector<11> &vec) {
    // vec indexed with 1-idx, vec[0] = 0
    11 n = vec.size() - 1;

    vector<11> ps(n+1, 0);
    for(11 i=1; i<=n; i++) {
        ps[i] = ps[i-1] + vec[i];
    }
    auto cost = [&](11 l, 11 r) {
        11 sum = ps[r] - ps[l-1];
        return sum * sum;
    };

    vector<11> cur(n+1, 0), nxt(n+1);

    // O(n log(n))
    function<void(11, 11, 11, 11)> compute = [&](11 l, 11 r, 11
        optl, 11 optr) {
        if (r < l) return;

        11 mid = (l+r)/2;

        11 best = INF;
        11 opt = -1;
        for(11 k=optl; k<=min(mid, optr); k++) {
            11 val = cur[k-1] + cost(k, mid);
```

```
        if (val < best) {
            best = val;
            opt = k;
        }
        nxt[mid] = best;

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

    for(11 i=1; i<=n; i++) // 1 partition
        cur[i] = cost(1, i);

    for(11 i=1; i<m; i++) { // m partitions
        nxt[0] = 0;
        compute(1, n, 1, n);
        swap(cur, nxt);
    }

    return cur[n];
}
```

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:

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

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:

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

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

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

$$C(a, c) + C(b, d) \leq C(a, d) + C(b, c)$$
 (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:**  $O(n^2)$

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

```
11 knuth(vector<11> &vec) {
    // vec indexed with 1-idx, vec[0] = 0
    11 n = vec.size() - 1;

    vector<11> ps(n+1, 0);
    for(11 i=1; i<=n; i++) {
        ps[i] = ps[i-1] + vec[i];
    }
    auto C = [&](11 l, 11 r) {
        return ps[r] - ps[l-1];
    };

    for(11 i=1; i<=n; i++) {
        opt[i][i] = i;
    }

    for(11 l=n-1; l>=1; l--) {
        for(11 r=l+1; r<=n; r++) {
            11 mn = INF;
            11 cost = C(l, r);
            for(11 k=opt[l][r-1]; k<=min(r-1, opt[l+1][r]); k
                ++){
                11 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:**  $O(\log N)$  per query and per insertion

```
// 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 11 a, b, p;
    bool operator<(const Line& o) const { return a < o.a; }
```



```
bool operator<(ll x) const { return p < x; }
};

struct DynamicCMT : 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->p;
    }
};
```

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  $x \geq t$ . In other words, once  $f(x)$  “win/lose”  $g(x)$ ,  $f(x)$  will continue to “win/lose”  $g(x)$ .

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 strictly increasing.

slope-trick.cpp

**Description:** Using Slope trick, compute the min cost to modify array 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  $f_{-i}(x) = \min(g_{-i}(t) \text{ pra } t \leq x)$ 

// podemos escrever  $g_i(x) = f_{i-1}(x) + |x-v[i]|$ 
// 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  $f_{i-1}$ )

// 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.

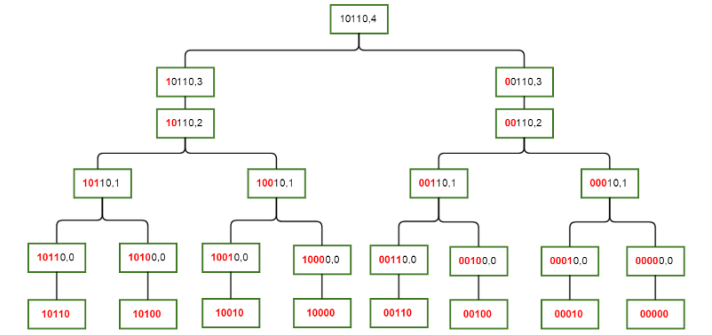
This can be modified for other operations in which the subset contributes for the mask . *Example:*

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:  $\mathcal{O}(2^N N)$ ,  $N$  = 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  $x \mid y = x$ 
// the number of elements y such that  $x \& y = x$ 
// the number of elements y such that  $x \& y \neq 0$ 

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;
    }
}
```

}

## 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  $s \neq 0$  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  $\geq 0$ .

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

$number_u = \text{MEX of all } number_{v_i}$

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 **guarantees** that all nim-values smaller than the considered nimber can be reached, which is essentially 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 numbers 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) { //  $\mathcal{O}(n \log(n))$ 
        for(ll i=0; i<=max_valid_value; i++)
            missing.insert(i);
    }

    ll get() { //  $\mathcal{O}(1)$ 
        if (missing.empty()) return 0;
    }
}
```

```
return *missing.begin();
}

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

void add(ll val) { //  $\mathcal{O}(\log(n))$ 
    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.

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://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:**  $\mathcal{O}(1)$

Te1lab, 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:**  $\mathcal{O}(N \log N)$

---

*// 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.

### manhattanMST dfs bfs dinic

- 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 for anonymous function

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

---

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

---

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

---

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

49775d, 88 lines

```

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

    vector<Edge> edges;
    vector<vector<ll>> g;
    vector<ll> level, ptr;

    // n need to be big enough for all nodes, including src/
        sink
    ll n, src, sink;
    Dinitz(ll nn, ll s = -1, ll t = -1) : n(nn+10) {
        src = (s == -1 ? n-2 : s);

```

```

        sink = (t == -1 ? n-1 : t);
        g.resize(n);
    }

    void addEdge(ll u, ll v, ll cap, ll rcap = 0) { // rcap =
        retrocapacity for bidirectional 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;
}
};

```

A maximum matching has the maximum cardinality. A perfect matching is a maximum matching. But the opposite is not necessarily true.

Let's define an s-t cut  $C = (S\text{-component}, T\text{-component})$  as a partition of  $V \in G$  such that source  $s \in S\text{-component}$  and sink  $t \in T\text{-component}$ . Let's also define a cut-set of  $C$  to be the set  $(u, v) \in E \mid u \in S\text{-component}, v \in T\text{-component}$  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:**  $\mathcal{O}(E \log(E))$

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 complementary 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

e51a5b, 88 lines

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

```
};

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

// n need to be big enough for all nodes, including src/
// sink
ll n, src, sink;
Dinitz(ll nn, ll s = -1, ll t = -1) : n(nn+10) {
    src = (s == -1 ? n-2 : s);
    sink = (t == -1 ? n-1 : t);
    g.resize(n);
}

void addEdge(ll u, ll v, ll cost, ll cap, ll rcap = 0) { //
    rcap = retrocapacity for bidirectional 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

### 6.4.1 k-Coloring

**TODO:** Add this blog

<https://codeforces.com/blog/entry/57496>

[https://en.wikipedia.org/wiki/Graph\\_coloring](https://en.wikipedia.org/wiki/Graph_coloring)

<https://open.kattis.com/problems/coloring>

## 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  $\leq d[u]$ , 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 discarded
            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:**  $\mathcal{O}(V * E) \rightarrow \mathcal{O}(N^2)$

```
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:**  $\mathcal{O}(V * E) \rightarrow \mathcal{O}(N^2)$

```
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());
    }
}
```

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 < sqrt(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 Undirected Graph

Bridges and Articulation Points are concepts for undirected graphs!

6.6.1 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  $low[v] \geq tin[u]$ .

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);
}
```

6.6.2 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]);
            }
        }
    }
}
```

6.6.3 Articulation Points

One Vertex in a graph is considered a Articulation Points or Cut Vertex 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);
}
```

6.6.4 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-conexas 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-vertice-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);
        }
    }
}
```

6.6.5 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

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

```
// 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;
}
```

6.7 Directed Graph

6.7.1 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 contradiction 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:**  $\mathcal{O}(V + E)$

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

void dfs(ll u) {
    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());
}
```

6.7.2 Kosaraju

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

```
// 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());
}
```

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

```
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_) { // 1-idx

        gi.assign(n+1, vll());
        for(ll i=1; i<=n; i++) {
            for(auto j : g[i])
                gi[j].pb(i);
        }

        gc.assign(n+1, vll());
        vis.assign(n+1, 0);
        comp.assign(n+1, -1);
        st = stack<ll, vll>();
    }
}
```

```

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]);
}
};

```

### 6.7.3 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);
}
};

```

## 6.8 Trees

#### 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};
}

};
```

Mathematics (7)

7.1 Modular Arithmetic

```

modular.cpp
Description: mint struct for modular arithmetic operations
Time: O(1) for most operations, O(log(n)) for division and exponentiation
// 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 fexp(Z x, ll i) {
        if (i == 0) return 1;
        if (i == 1) return x;
        Z m = fexp(x, i/2);
        m *= m;
        if (i & 1) return m * x;
        else return m;
    }

    Z& operator /=(Z rhs) {
        return *this *= fexp(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  $p=3$ )  $= 1 \cdot 3^2 + 0 \cdot 3^1 + 2 \cdot 3^0$   
 $\implies n_2 = 1, n_1 = 0, n_0 = 2$

7.2 Combinatorics

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}, \quad 0 \leq m \leq n$$
  
0, 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: O(n) to construct, O(1) operations
// remeber to import mint struct !!
struct Combinatorics {
    vector<mint> fact, ifact;

    Combinatorics(ll n) : fact(n+1), ifact(n+1) { // inclusive
        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;
    }

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

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

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

    // "Arranjo Simples"
```

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

// "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 starsBars(ll n, ll k) {
    if (k == 0) {
        if (n == 0) return 1;
        else return 0;
    }
    return combination(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  $G$ .

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

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

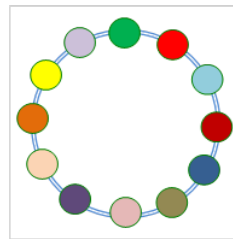
$$\text{orb}(x) = \{y \in X : \exists g \in G \text{ } gx = y\}$$

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

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

**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.

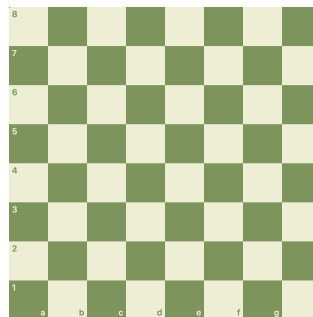
### fft-simple



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

**Example2:** Count the number of different  $n \times n$  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.



$$G(\text{Rotations}) = 0^\circ, 90^\circ, 180^\circ, 270^\circ$$

$$f(\text{rotation}) =$$

$$\begin{aligned} 0^\circ : 2^{(n^2)} \\ 90^\circ / 270^\circ : 2^{\frac{n^2}{4}}, & \quad n_{\text{even}} \\ & 2^{\frac{n^2-1}{4}} \cdot 2, & \quad n_{\text{odd}} \\ 180^\circ : 2^{\frac{n^2}{2}}, & \quad n_{\text{even}} \\ & 2^{\frac{n^2-1}{2}} \cdot 2, & \quad n_{\text{odd}} \end{aligned}$$

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

### 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:*

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

## 7.3 FFT

FFT can be used to turn a polynomial multiplication complexity to  $O(N \log N)$ .

A **convolution** 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:**  $O(N \log N)$

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;
}
```

Number theory (8)

8.1 Sieves

These sieves are used to find all primes up to an upper bound N, which is usually 10<sup>7</sup>

8.1.1 Eratosthenes

Eratosthenes uses less memory than the linear sieve and is almost as fast

```
eratosthenes.cpp
Description: Optimized sieve of eratosthenes
Time: O(N log log N)
// 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 eratosthenes(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;
        }
    }
}

8.1.2 Linear Sieve
Due to the lp vector, one can compute the factorization of any number very quickly!

Can check primality with lp[i] == i

Uses more memory, because lp is a vector of int or ll and not bits.

Proof of time complexity:
We need to prove that the algorithm sets all values lp[] correctly, and that every value will be set exactly once. Hence, the algorithm will have linear runtime, since all the remaining actions of the algorithm, obviously, work for O(n).

Notice that every number i has exactly one representation in form:

i = lp[i] · x,

where lp[i] is the minimal prime factor of i, and the number x doesn't have any prime factors less than lp[i], i.e.

lp[i] ≤ lp[x].

Now, let's compare this with the actions of our algorithm: in fact, for every x it goes through all prime numbers it could be multiplied by, i.e. all prime numbers up to lp[x] inclusive, in order to get the numbers in the form given above.

Hence, the algorithm will go through every composite number exactly once, setting the correct values lp[] there. Q.E.D.

linear-sieve.cpp
Description: Linear Sieve that iterates every value once (prime) or twice (composite)
Time: O(N)
vector<ll> primes, lp(MAX);
// lp[i] = smallest prime divisor of i

void linearSieve(ll n) {
    for (ll i=2; i <= n; i++) {
        if (lp[i] == 0) { // i is prime
            lp[i] = i; // {lp[i] == i} for prime numbers
            primes.pb(i);
        }
        // visit every composite number that has primes[j] as the lp
        for (ll j = 0; i * primes[j] <= n; j++) {
            lp[i * primes[j]] = primes[j];

            if (primes[j] == lp[i])
                break;
        }
    }
}
```

8.2 Extended Euclid

Solves the  $ax + by = gcd(a, b)$  equation.

8.2.1 Inverse Multiplicative

if  $gcd(a, b) = 1$  :

then:

$ax + by \equiv 1$

also, if you apply (mod 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  $gcd(a, b) = 1$

8.2.2 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: O(log min(a, b))
// 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) = pll{x1, x - q * x1};
        tie(y, y1) = pll{y1, y - q * y1};
        tie(a1, b1) = pll{b1, a1 - q * b1};
    }
    return a1;
}
```



# Strings (9)

## 9.1 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 = {
        1000000009,10000000021,10000000033,
        10000000087,10000000093,10000000097,
        1000000103,1000000123,1000000181,
        1000000207,1000000223,1000000241,
        1000000271,1000000289,1000000297
    };

    shuffle(mods.begin(), mods.end(), rng);

    string s; cin >> s;

    Hashing hash(s, mods[0]);
}
```

## 9.2 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  $ab cab ab cab$  are  $ab$  and  $ab cab$ . ( $z[8] = 2$ ,  $z[5] = 5$ ) ( $z[i] = n-i$ )
- 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  $ab cab ca$  are **abc**, **abcabc** and **abcabca**.

It works because  $(z[i] + i \leq n)$  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:

$abaabababab\ z[8] = 2$

**abaababab** 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) { //  $\mathcal{O}(n)$ 
    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;
}
```

## 9.3 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  $\pi$  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  $"abcab cd"$  is  $[0, 0, 0, 1, 2, 3, 0]$ , and prefix function of string  $"aabaab"$  is  $[0, 1, 0, 1, 2, 3]$ .

```
kmp.cpp
Description: Computes the prefix function
Time:  $\mathcal{O}(n)$ 
3f2929, 13 lines

vector<ll> kmp(string &s) { //  $\mathcal{O}(n)$ 
    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;
}
```

### 9.3.1 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.

## 9.4 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 substrng between  $s.substr(sa[i])$  and  $s.substr(sa[i+1])$ , which are the substrings corresponding to the common 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;
}
```

9.5 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);
};
```

9.6 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;
}
```

9.7 Suffix Automaton

The goat!!!

9.7.1 Concepts:

- 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 positions (of the last character) in which there is an occurrence 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 constraints, 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 a 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 contains a bigger subset *endpos(link(u))*, that contains all positions 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  $O(3n)$ .

### 9.7.2 Implementation:

The implementation can be changed to use a map instead of a fixed vector for adjacent edges. This will increase the time complexity to  $O(n \log k + constant \cdot map)$  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:**  $O(n)$  to create all nodes,  $O(n \log n)$  to compute endpos size

```
// 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 (10)

### 10.1 Random Generator

random.cpp

**Description:** Good randomizer to generate int in a range or shuffle vectors

**Time:**  $\mathcal{O}(1)$  for randint,  $\mathcal{O}(n \log(n))$  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;
}
```

### 10.2 Read an Fraction Input

```
char c;
ll num, den;
cin >> num >> c >> den;
```

### 10.3 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
}
```

10.4 Bit optimization

use popcnt pragma!!

#pragma GCC target ("popcnt")

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

- **\_\_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

10.4.2 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\_ulong()**: 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;

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.

10.4.3 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  $__builtin_popcount(a \oplus b)$  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.

# Techniques (A)

techniques.txt	160 lines
Recursion	
Divide and conquer	
Finding interesting points in N log N	
Algorithm analysis	
Master theorem	
Amortized time complexity	
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	
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	
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	
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	
Tries	
Rolling polynomial hashes	
Suffix array	
Suffix tree	
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	