



Universidade de Brasília

Tomate Cerveja

Wallace Wu, Pedro Gallo, Henrique Ramos

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

Contest (1)

template.cpp39 lines

```
#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))
#define LSB(i) ((i) & ~(i))
#define MSB(i) (32 - __builtin_clz(i)) //64 - clzll
#define BITS(i) __builtin_popcountll(i) //count set bits

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

#define debug(a...) cerr<<#a<<" ";for(auto b:a)cerr<<b<<" ";
cerr<<endl;
template<typename... A> void dbg(A const&... a){(cerr<<"{"<<a
<<"} ", ...);cerr<<endl;}

const ll MAX = 3e5+10;
const ll MOD = 1e9+7;
const ll INF = 0x3f3f3f3f3f3f3f3f;
const ll LLINF = INT64_MAX;
const ld EPS = 1e-7;
const ld PI = acos(-1);
```

1#include <chrono>
using namespace std::chrono;
int32_t main(){ sws;
auto start = high_resolution_clock::now();
// function here
auto stop = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(stop - start);
cout << duration.count() << endl;
}

.bashrc1 lines

alias comp='g++ -std=c++17 -g3 -ggdb3 -O3 -Wall -Wextra -
fsanitize=address,undefined -Wshadow -Wconversion -
D_GLIBCXX_ASSERTIONS -o test'

hash.sh3 lines

Hashes a file, ignoring all whitespace and comments. Use for
verifying that code was correctly typed. CTRL+D to send EOF
cpp -dD -P -fpreprocessed | tr -d '[:space:]' | md5sum | cut -c
-6

troubleshoot.txt52 lines

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

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

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

Time limit exceeded:
Do you have any possible infinite loops?

What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

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

Mathematics (2)

2.1 FFT

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

A **convolution** is easily computed by inverting one of the vector and doing the polynomial multiplication normally.

fft-simple.cppDescription: Computes the product between two polynomials using fftTime: O(N log N)24d5df, 70 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;
        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;
}
```

Data structures (3)

3.1 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>;
```

3.2 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, $O(\alpha) \approx O(5)$

```
dsu.cpp
Description: Disjoint Set Union with path compression and tree balancing
Time: O(α) 0479c4, 22 lines

struct DSU{
    vll group, card;
    DSU (ll n){
        n += 1; // 0-idx -> 1-idx
        group = vll(n);
        iota(group.begin(), group.end(), 0);
        card = vll(n, 1);
    }
    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;
    }
};
```

3.3 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: 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];
};
```

3.4 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.

3.4.1 Recursive SegTree

```
segRecursive.cpp
Description: Basic Recursive Segment Tree for points increase and range sum query. When initializing the segmente tree, remeber to choose the range limits (L, R)
Time: O(N log N) to build, O(log N) to increase or query 156cd2, 70 lines

// [0, n] segtree for range sum query, point increase
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 operaton:
            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)
        );
    }
}
};

```

3.4.2 PA Segtree

segPA.cpp

Description: Seg with PA (Progressao Aritmetica / Arithmetic Progression)
When initializing the segmente tree, remeber 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 22f4a0, 100 lines

// [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)
        );
    }
}
};

```

Dynamic Programming (4)

4.1 Convex Hull Trick

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

Some valid functions:

$ax + b$

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

cht-dynamic.cpp

Description: Dynamic version of CHT, thefore, one can insert lines in any order. There is no line removal operator

Time: $\mathcal{O}(\log N)$ per query and per insertion

707da4, 51 lines

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return 1.a * x + 1.b;
}
};
```

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_it. In other words, once f(x) “win/lose” g(x), f(x) will continue to “win/lose” g(x).

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

Game theory (5)

5.1 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.

5.2 Bouton’s Theorem

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

5.2.1 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)

5.3 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.

5.4 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 *i* 0.

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

number_u = 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:

5.4.1 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.

5.4.2 Decomposition

MEX operator to compute the number 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: O (log N) per addition/removal, O (1) to get mex value, O (N log(N)) to initialize
d6f2b9, 27 lines

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

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

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

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

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

5.5 Variations and Extensions

5.5.1 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 number 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.

5.6 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.

5.7 Staircase Nim

5.7.1 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.

5.7.2 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.

5.8 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.

5.8.1 Degenerate (Base) States

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

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

5.8.2 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 $1e6$).

5.9 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>

5.10 References

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

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

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

Numerical (6)

Number theory (7)

7.1 Sieves

These sieves are used to find all primes up to an upper bound N , which is usually 10^7

7.1.1 Eratosthenes

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

```
eratosthenes.cpp
Description: Optimized sieve of eratosthenes
Time:  $\mathcal{O}(N \log \log N)$ 
8d74e5, 15 lines

//  $\mathcal{O}(N \log^2(N)) \rightarrow$  Teorema de Merten
vector<ll> primes {2, 3};
bitset<MAX> sieve; // {sieve[i] == 1} if i is prime
// MAX can be  $\sim 1e7$ 

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;
        }
    }
}
```

7.1.2 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 $\mathcal{O}(n)$.

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

$$i = lp[i] \cdot x,$$

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.

$$lp[i] \leq lp[x].$$

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: $\mathcal{O}(N)$

2124a6, 18 lines

```
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;
        }
    }
}
```

7.2 Extended Euclid

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

7.2.1 Inverse Multiplicative

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

then:

$$ax + by \equiv 1$$

also, if you apply $\pmod b$ to the equation:

$$ax \pmod b + by \pmod b \equiv 1 \pmod b$$

$$ax \equiv 1 \pmod b$$

In other words, one can find the inverse multiplicative of any number a in modulo b if $gcd(a, b) = 1$

7.2.2 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: $\mathcal{O}(\log \min(a, b))$

60dd70, 16 lines

```
// equation: a*x + b*y = gcd(a, b)
// input: (a, b)
// returns gcd of (a, b)
// also computes &lx and &ly, 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;
}
```

Combinatorial (8)

8.1 Permutations

8.1.1 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		

Graph (9)

9.1 Network flow

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

dinic.cpp

Description: Run several bfs to compute the residual graph until a max flow configuration is discovered

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

dea1b7, 86 lines

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

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

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

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

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

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

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

```

}

void reset_flow() {
    for (ll i=0; i<n; 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;
}
};

```

9.1.1 Minimum Cut

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

Let's define an s-t cut $C = (S\text{-component}, T\text{-component})$ 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 $S\text{-component}$. All other unreachable vertices belong to the $T\text{-component}$. All edges connecting the $S\text{-component}$ to the $T\text{-component}$ 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.

9.1.2 Matching with Flow

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

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

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

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

9.1.3 Maximum Independent Set

TODO: Add this blog <https://ali-ibrahim137.github.io/competitive/programming/2020/01/02/>

maximum-independent-set-in-bipartite-graphs.
html

9.2 Matching

9.2.1 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: $O(n^3)$

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 mínimo, 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;
}
}
```

9.3 Coloring

9.3.1 k-Coloring

TODO: Add this blog

https://codeforces.com/blog/entry/57496
https://en.wikipedia.org/wiki/Graph_coloring
https://open.kattis.com/problems/coloring

9.4 Shortest Paths

For weighted directed graphs

9.4.1 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, transverse only using the smallest distances.
Time: O((V + E) log V)
ef1d0b, 19 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 [p1, u] = pq.top(); pq.pop();
        if (p1 > d[u]) continue;
        for(auto [v, p2] : g[u]){
            if (d[u] + p2 < d[v]){
                d[v] = d[u] + p2;
                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: 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);
            }
        }
    }
}
```

```
bellman-ford.cpp
Description: n-1 iterations is sufficient to find all shortest paths
Time: O(V * E) -> 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: O(V * E) -> 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());
        }
    }
}
```

9.4.3 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: 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]);
}
```

9.5 Undirected Graph

Bridges and Articulation Points are concepts for undirected graphs!

9.5.1 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 $low[v] \nless$ $tin[u]$.

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

```
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);
}
```

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

```
// 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]);
            }
        }
    }
}
```

9.5.3 Articulation Points

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

articulation.cpp

Description: if $low[v] \geq tin[u]$, u is an articulation points The root is a corner case
Time: $\mathcal{O}(V + E)$

```
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);
}
```

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

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

9.5.5 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

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

206ba3, 21 lines

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

set<array<ll, 3>> edges;

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

9.6 Directed Graph

9.6.1 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:  $\mathcal{O}(V + E)$ 
75f781, 17 lines

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

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

// P-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());
}

```

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

```

// P-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:  $\mathcal{O}(V + E)$ 
381904, 45 lines

```

```

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

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

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

    Kosaraju(vector<vll> &g_)
        : g(g_), n(g_.size()-1) { // 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]);
}
};

```

9.6.3 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);
}

};

```

9.7 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
}
};

```

queryTree.cpp

Description: Binary Lifting for min, max weight present in a simple path

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};
}
};

```

9.8 Math

Geometry (10)

Strings (11)

11.1 Hashing

Hashing consists in generating a Polynomial for the string, therefore, assigning each distinct string to a specific numeric value. In practice, there will always be some collisions:

Probability of collision: $= \frac{n^2}{2m}$

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]);
}

```

11.2 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 simple problems:

- Find all occurrences of a pattern p in another string s . ($p + 's' + s$) ($z[i] == p.size()$)

- 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 `abcbabcbab` are `ab` and `abcb`. ($z[8] = 2$, $z[5] = 5$) ($z[i] = n-i$)
- 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 `abcbabca` are **abc**, **abcbabc** and **abcbabca**.

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

`abaababab` $z[8] = 2$

abaababa is the period; the remaining ($z[i]$ characters) are a prefix of the period; and when all these characters are combined, it can form the string (which has n characters).

zfunction.cpp
Description: For each substring starting at position i , compute the maximum match with the original prefix. $z[0] = 0$
Time: $\mathcal{O}(n)$

```
vector<ll> z_function(string &s) { // O(n)
    ll n = (ll) s.length();
    vector<ll> z(n);
    for (ll i=1, l=0, r=0; i<n; i++) {
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);

        while (i + z[i] < n and s[z[i]] == s[i + z[i]]) z[i]++;

        if (r < i + z[i] - 1) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

11.3 KMP

KMP stands for Knuth-Morris-Pratt and computes the prefix function.

You are given a string s of length n . The prefix function for this string is defined as an array π of length n , where $\pi[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $\pi[0] = 0$.

For example, prefix function of string `"abcbabcd"` 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: $\mathcal{O}(n)$

```
vector<ll> kmp(string &s) { // O(n)
    ll n = (ll) s.length();
```

```
vector<ll> pi(n);
for (ll i = 1; i < n; i++) {
    ll j = pi[i-1];
    while (j > 0 && s[i] != s[j])
        j = pi[j-1];
    if (s[i] == s[j])
        j++;
    pi[i] = j;
}
return pi;
}
```

11.4 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
1 anana
2 nana
3 ana
4 na
5 a

Sort the Suffixes
----->
alphabetically

5 a
3 ana
1 anana
0 banana
4 na
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: $\mathcal{O}(N \log N)$

```
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] ? 1 : 0;
        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: $\mathcal{O}(N \log N)$

```
vector<ll> kasai(string s, vector<ll> sa) {
    ll n = s.size(), k = 0;
    vector<ll> ra(n), lcp(n);
    for (ll i = 0; i < n; i++) ra[sa[i]] = i;

    for (ll i = 0; i < n; i++, k -= !!k) {
        if (ra[i] == n-1) { k = 0; continue; }
        ll j = sa[ra[i]+1];
        while (i+k < n and j+k < n and s[i+k] == s[j+k]) k++;
        lcp[ra[i]] = k;
    }
    return lcp;
}
```

Problems that can be solved:

Numbers of Distinct Substrings:

- $\frac{n(n+1)}{2} - lcp[i]$ (for all i)

Longest Repeated Substring:

- biggest $lcp[i]$. The position can be found in $sa[i]$

Find how many distinct substrings there are for each len in $[1:n]$:

- Use delta encoding and the fact that $lcp[i]$ counts the repeated substring between $s.substr(sa[i])$ and $s.substr(sa[i+1])$, which are the substrings corresponding to the commom prefix.

Find the k -th distinct substrng:

```
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;
}
```

11.5 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);
    }
};
```

11.6 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) { //  $\mathcal{O}(n)$ 
```

```
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;
}
```

Miscellaneous (12)

12.1 Ternary Search

ternary-search.cpp
Description: Computes the min/max for a function that is monotonically increasing then decreasing or decreasing then increasing.
Time: $\mathcal{O}(N \log N_3)$

c3a5d7, 48 lines

```
/*
Float and Min Version: Requires EPS (precision usually defined
in the question text)
*/

ld f(ld d){
    // function here
}

// for min value
ld ternary_search(ld l, ld r){
    while(r - l > EPS){
        // divide into 3 equal parts and eliminate one side
        ld m1 = l + (r - l) / 3;
        ld m2 = r - (r - l) / 3;
        if (f(m1) < f(m2)){
            r = m2;
        }
        else {
            l = m1;
        }
    }
    return f(l); // check here for min/max
}

/*
Integer and Max Version:
*/

ll f(ll idx) {
```

```
    // function here
}

// for max value, using integer idx
ll ternary_search(ll l, ll r) {
    while(l <= r) {
        // divide into 3 equal parts and eliminate one side
        ll m1 = l + (r-l)/3;
        ll m2 = r - (r-l)/3;
        if(f(m1) < f(m2)) {
            l = m1+1;
        }
        else {
            r = m2-1;
        }
    }
    return f(l); // check here for min/max
}
```

Techniques (A)

techniques.txt	159 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	
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