Compilation

```
FLAGS=-Wall -Wextra -Wshadow -Wno-unused-result -D_GLIBCXX_DEBUG -fsanitize=address
↪  -fsanitize=undefined -fno-sanitize-recover

run_a: clean
        @g++ a.cpp $(FLAGS) -D_LOCAL_ && ./a.out < d.in
```

Template

```
#include <bits/stdc++.h>
using namespace std;
ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);

cout.precision(20); cout << fixed << ans;
n digits = floor(log10(num)) + 1;
```

Formulas

Catalan number: $nCr(2n, n)/(n + 1)$

Bitmask

```
//iterate over all subsets of a set
//0, lsb, 2lsb, 2lsb & lsb, 3lsb...
do{
    } while((b = (b - x) & x));

//m will be all submask of m, consider 0 appart
for(m = n; m > 0; m = (m-1)&n) {}
```

Point

```
struct Point{
    ll x, y, w; //weight
    bool operator == (Point a) {
        return x == a.x && y == a.y;
    }
    bool operator != (Point a) {
        return x != a.x || y != a.y;
    }
    bool operator < (Point a) {
        if(x != a.x) return x < a.x;
        return y < a.y;
    }
    ll getquad() {
        if(x > 0 && y >= 0) return 1;
        if(x <= 0 && y > 0) return 2;
        if(x < 0 && y <= 0) return 3;
        if(x >= 0 && y < 0) return 4;

        return 0;
    }
    void show() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
const int LEFT = 1;
const int RIGHT = -1;
int orientation(Point p1, Point p2, Point p3){
    ll ans = (p2.x - p1.x) * (p3.y - p2.y) - (p3.x - p2.x) * (p2.y - p1.y);
    if(ans == 0) return ans;
    return ans > 0 ? LEFT : RIGHT;
}
```

```
//Sort counterclockwise the points, with the center in v[0].
//Starts at 6:00, no 3 points are collinear
bool cmp_counterclockwise(Point a, Point b) {
    if(a.x == v[0].x) return true;
    if(b.x == v[0].x) return false;
    if(a.x > v[0].x && v[0].x > b.x) {
        return true;
    }
    if(b.x > v[0].x && v[0].x > a.x) {
        return false;
    }
    return orientation(a, v[0], b) == RIGHT;
}


point firstPoint;
//Graham compare
bool compare(point a, point b){
    if(a == firstPoint) return true;
    if(b == firstPoint) return false;
    int o = orientation(firstPoint, a, b);
    if(o == 0) return a.x < b.x;
    return o == LEFT? true : false;
}

//supposse n > 2
ll szHull;
vector<point> Hull;
//Graham Scan, O(nlogn)
void convexHull(vector<point> &vpoint, int n){
    int i, o;
    firstPoint = vpoint[0];
    for(i = 1; i < n; ++i) ckmin(firstPoint, vpoint[i]);

    sort(vpoint.begin(), vpoint.end(), compare);

    if(n <= 2) {cout << "0"; exit(0);}

    Hull = {vpoint[0], vpoint[1]};
    szHull = 2;

    for(i = 2; i < n; ++i){
        o = orientation(Hull[szHull - 2], Hull[szHull - 1], vpoint[i]);
        while(o == RIGHT || o == 0) {
            Hull.pop_back();
            --szHull;
            if(szHull < 2) break;
            o = orientation(Hull[szHull - 2], Hull[szHull - 1], vpoint[i]);
        }
        Hull.pb(vpoint[i]);
        ++szHull;
    }

}
```

Graphs

```
//shows in O(n) SCC Strongly Connected Components.
//first dfs1 when node is processed is added to the stack, then the graph is transposed, then dfs2
↪  when node is processed is showed in its component
vector<vi> graph, graphRev;
stack<int> s;
vector<bool> visited;
ll n;
```

```
void dfs1(ll u){
    visited[u] = true;
    for(auto v : graph[u]){
        if(visited[v] == false) dfs1(v);
    }
    s.push(u);
}
void dfs2(ll u){
    visited[u] = true;
    for(auto v : graphRev[u]){
        if(visited[v] == false) dfs2(v);
    }
    cout << u << ".";
}

void Kosaraju(){
    ll i, j;
    //transpose graph to graphRev
    for(i = 0; i < n; ++i){
        for(j = 0; j < sz(graph[i]); ++j){
            graphRev[graph[i][j]].pb(i);
        }
    }
    visited.assign(n, false);
    dfs1(0);
    visited.assign(n, false);

    while(true) {
        while(s.empty() == false && visited[s.top()] == true) s.pop();
        if(s.empty() == true) break;
        dfs2(s.top());
        cout << endl;
    }
}

//Bellman Ford, try to relax all edges n times. If you can still relax 1 edge more then there are a
↪  negative cycle

//LCA in a tree rooted at 0
int parent[LOG_N][MAX_N]; //parent[i][j] is the ancestor 2^i of node j. Is a sparse table
int level[MAX_N]; //depth of the node in the tree

//call dfs0(0, 0);
void dfs0(int u, int p) {
    parent[0][u] = p;
    for(auto v : graph[u]) {
        if(v == p) continue;
        level[v] = level[u] + 1;
        dfs0(v, u);
    }
}


//O(n log n)
void preprocess() {
    int i, j;
    dfs0(0, 0);
    for(i = 1; i < LOG_N; ++i) {
        for(j = 0; j < MAX_N; ++j) {
            parent[i][j] = parent[i - 1][parent[i - 1][j]];
        }
    }
}
```

```
//rise b to the same level as a and continue moving up. O(log n)
int lca(int a, int b) {
    int i;

    if(level[a] > level[b]) swap(a, b);
    int d = level[b] - level[a];

    for(i = 0; i < LOG_N; ++i) {
        if((d >> i) & 1) b = parent[i][b];
    }
    if(a == b) return a;

    for(i = LOG_N - 1; i >= 0; --i) {
        if(parent[i][a] != parent[i][b])
            a = parent[i][a], b = parent[i][b];
    }

    return parent[0][a];
}

//distance between two nodes in a tree
int dist(int u, int v) {
    return level[u] + level[v] - 2 * level[lca(u, v)];
}
//standard Ford Fulkerson algorithm for maxFlow.
//Need a ady matrix graph[][]
vll parent;
bool BFS() {
    ll u, v;
    vector<bool> visited(n + 2, false);
    queue<ll> q;
    q.push(n);
    visited[n] = true;
    while(q.empty() == false) {
        u = q.front();
        q.pop();

        for(v = 0; v < n + 2; ++v) {
            if(visited[v] == false && graph[u][v] > 0) {
                parent[v] = u;
                visited[v] = true;
                if(v == n + 1) return true;
                q.push(v);
            }
        }
    }
    return false;
}
ll maxFlow() { //0-1 maxFlow, doesn't need to check for m's values
    ll max_flow = 0, v, m;
    parent.assign(n + 2, -1);
    while(BFS()) {
        m = LLONG_MAX;
        for(v = n + 1; v != n; v = parent[v]) {
            m = min(m, graph[parent[v]][v]);
        }
        max_flow += m;
        for(v = n + 1; v != n; v = parent[v]) {
            graph[parent[v]][v] -= m;
            graph[v][parent[v]] += m;
        }
    }

    return max_flow;
}
```

Totient function + Linear Sieve

```
const ll NUM_MAX = 1e5 + 10;
bool num[NUM_MAX];//if num[i] = false => i is prime
vector<int> prime;
ll phi[NUM_MAX];
//phi[x] is the number of coprime numbers with x. phi(p^k) = p^k - p^(k-1)
void totient_function_generator(int max_num_primo){ //Works in O(n), also creates a sieve.
    int i, j, prime_size = 0;
    phi[1] = 1;

    for(i = 2; i <= max_num_primo; ++i){
        if(num[i] == false) {prime.push_back(i); ++prime_size; phi[i] = i - 1;} //If p is prime
        for(j = 0; j < prime_size && i * prime[j] <= max_num_primo; ++j){
            num[i * prime[j]] = true;
            if(i % prime[j] != 0)
                phi[i * prime[j]] = phi[i] * phi[prime[j]]; //If p doesn't divide i
            else {
                phi[i * prime[j]] = prime[j] * phi[i];
                break;
            } //If p divides i
        }
    }
}
```

Mod and combinatorics

```
//cout << a << "*(" << x << ") + " << b << "*(" << y << ") = " << g << endl;
ll gcdEx(ll a, ll b, ll *x1, ll *y1) {
    if(a == 0) {
        *x1 = 0;
        *y1 = 1;
        return b;
    }
    ll x0, y0, g;
    g = gcdEx(b%a, a, &x0, &y0);
    *x1 = y0 - (b/a)*x0;
    *y1 = x0;
    return g;
}
//Inverse with mod
ll elevate(ll a, ll b){
    ll ans = 1;
    while(b){
        if(b & 1) ans = ans * a % mod;
        b >>= 1;
        a = a * a % mod;
    }
    return ans;
}
//a^(mod - 1) = 1, Euler
ll inv(ll a){
    return elevate(a, mod - 2);
}

//n choose r, nCr = n!/(r!*(n-r)!)
ll nCr(ll n, ll r){
    ll i, ans = 1;
    if(r > n/2) r = n - r; //Symmetry
    for(i = 1; i <= r; ++i){
        ans *= n -i + 1;
        ans /= i;
    }
    return ans;
}
```

```
//nCr with mod
ll comb(ll a, ll b) {
    if(b > a) return -1;
    return ((fact[a] * mfact[b])%mod) * mfact[a-b] %mod;
}
```

BIT+ k-th element

```
vll bit;
ll bit_n;
void ini(ll n) {bit_n = n; bit.assign(bit_n + 1, 0);}
void update(ll i, ll x) {for(i++; i <= bit_n; i += LSB(i)) bit[i] += x;}
ll query_sum(ll i) {ll sum = 0; for(i++; i > 0; i -= LSB(i)) sum += bit[i]; return sum;}
/*k-th smallest element set in the bit*/
ll k_element(ll k) {
    ll l = 0, r = bit_n -1, mid, sum;
    while(l < r) {
        mid = (l + r)/2;
        sum = query_sum(mid);
        if(sum >= k) r = mid;
        else l = mid + 1;
    }
    return r;
}
```

Z-algorithm

```
void z_array(string s){
    int i, L = 0,R = 0, n = s.length();
    vi z(n, 0);

    //the box is [L, R]
    for(i = 1; i < n; ++i){
        z[i] = max(0, min(z[i - L], R - i));
        while(i + z[i] < n && s[z[i]] == s[i + z[i]]){
            L = i; R = i + z[i]; ++z[i];
        }
    }
    echo(z);
}
```

KMP for pattern searching

```
//lps[i] : longest proper prefix in pat[0..i] that is also a suffix in pat[0...i]
void KMPsearch(string &txt, string &pat) {
    ll i, j, len = 0, n = txt.length(), m = pat.length();
    vi lps(m, 0);
    //create lps array
    for(i = 1; i < m;) {
        if(pat[i] == pat[len]) {
            ++len;
            lps[i++] = len;
        }
        else{
            if(len == 0) ++i;
            else len = lps[len - 1];
        }
    }
    //search for pat in txt
    for(i = 0, j = 0; i < n;) {
        if(txt[i] == pat[j]) {
            ++i;
            ++j;
            if(j == m) {
                cout << "FOUND AT:" << i - m << "\n";
                j = lps[j - 1];
            }
        }
        else {
            if(j == 0) ++i;
            else j = lps[j - 1];
        }
    }

    cout << "Finished" << endl;
}
```

Aho Corasik: Trie + Automaton

```
//construct trie O(m) + automaton O(mk), O(mk) memory, m = sum(len(word_i))
#define next asdfa
//size of alphabet, 26 lowercase
const int k = 26;

struct vertex{
    vi next;
    //number of words ending at current vertex
    int leaf;
    //ancestor p and ch is the transition of p->v
    int p;
    char pch;
    //proper suffix link of the vertex
    int link;
    vi go;
    //how many suffixes there are in the tree;
    int count;

    vertex(int _p, char _pch) {
        next.assign(k, -1);
        leaf = 0;
        this->p = _p;
        this->pch = _pch;
        link = -1;
        go.assign(k, -1);
        count = -1;
    }
};

vector<vertex> t = {{-1, '$'}};
int t_size = 1;

//add string to the trie t
void add_string(string s) {
    int c, p = 0;
    for(char ch : s) {
        c = ch - 'a';
        if(t[p].next[c] == -1) {
            t.pb({p, ch});
            t[p].next[c] = t_size++;
        }
        p = t[p].next[c];
    }
    t[p].leaf++;
}

//Search for any proper suffix of v that has next[c] transition
//call go(v, ch) for move the automaton from the vertex v using transition ch
int go(int v, char ch);
```

```
//get the proper suffix link of v. Once called, don't call anymore add_strings
int get_link(int v) {
    if(t[v].link == -1) {
        if(v == 0 || t[v].p == 0) t[v].link = 0;
        else t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}


int go(int v, char ch) {
    int c = ch - 'a';
    if(t[v].go[c] == -1) {
        if(t[v].next[c] != -1) t[v].go[c] = t[v].next[c];
        //The root doesn't have next[c]
        else if(v == 0) t[v].go[c] = 0;
        else {
            t[v].go[c] = go(get_link(v), ch);
        }
    }
    return t[v].go[c];
}

//get the count of v
int count(int v) {
    if(t[v].count == -1) {
        t[v].count = t[v].leaf;
        if(v != 0) t[v].count += count(get_link(v));
    }
    return t[v].count;
}

//search the number of the strings in the automaton that are in the text
int search_num_string(string text) {
    int p = 0, ans=0;

    for(auto ch : text) {
        ans += count(p);
        p = go(p, ch);
    }
    ans += count(p);
    return ans;
}
```

Suffix Automaton

```
#define next _42_
//Suffix Automaton, save a directed acyclic graph and a suffix link tree with all the suffix of a word
struct state {
    //length of the longest string in the equivalence classes
    int len;
    //suffix link
    int link = -1;
    map<char, int> next;
    state(int _len) {
        len = _len;
    }
};

vector<state> t = {{0}};
int t_size = 1, last = 0;

//add a character to the automaton
//last is the state of the last char c added, p is the head of the automaton
```

```
//q is the state to duplicate
void sa_extend(char c) {
    int p = last, q;
    t.pb({t[last].len + 1});
    last = t_size; t_size++;
    //add c to the previous suffixes
    while(p != -1 && t[p].next.find(c) == t[p].next.end()) {
        t[p].next[c] = last;
        p = t[p].link;
    }
    //first time of c in the string
    if(p == -1) {
        t[last].link = 0;
        return;
    }
    q = t[p].next[c];
    if(t[p].len + 1 == t[q].len) {
        t[last].link = q;
        return;
    }
    //clone state q
    t.pb({t[p].len + 1});
    t_size++;
    t[t_size - 1].next = t[q].next;
    t[t_size - 1].link = t[q].link;

    //add links of last and q
    t[last].link = t_size - 1;
    t[q].link = t_size - 1;

    //point the last suffixes to q cloned
    while(p != -1 && t[p].next.find(c) != t[p].next.end()) {
        t[p].next[c] = t_size - 1;
        p = t[p].link;
    }
}
//O(s.length()) to create the automaton. Be careful adding any char once called another function
void sa_ini(string s) {
    for(char c : s) sa_extend(c);
}

//A path from root to a terminal node is a suffix of the automaton string
vector<bool> terminal;
void sa_terminal() {
    int p = last;
    if(terminal.empty() == false) return; //previously calculated
    terminal.assign(t_size, false);
    while(p != -1) {
        terminal[p] = true;
        p = t[p].link;
    }
}
//true if w is a substring of the automaton string
//Also s is the longest prefix of w that is in s
//w is a suffix if the last p is a terminal state
bool sa_is_substr(string w) {
    int p = 0; //string s;
    for(char ch : w) {
        if(t[p].next.find(ch) == t[p].next.end()) return false;
        p = t[p].next[ch];
        //s += c;
    }
    return true;
}
```

```
vll dp_num_substr;
ll num_substr_rec(int i) {
    ll sum = 1;
    if(dp_num_substr[i] != -1) return dp_num_substr[i];
    for(auto el : t[i].next) sum += num_substr_rec(el.se);
    return dp_num_substr[i] = sum;
}
//Number of different substrings of the automaton string (Is the number of different paths in the automaton)
//For the number of the length of all different substring the recursive formula is
// sum of dp_num_substr[i] + dp_num_len_substr[i], the previous answer + 1*number of different substrings
ll sa_num_substr() {
    if(dp_num_substr.empty() == false) return dp_num_substr[0]; //previously calculated
    dp_num_substr.assign(t_size, -1);
    num_substr_rec(0);
    return dp_num_substr[0]; // -1 if you don't want the empty substring
}

//k-th string in the sorted substrings set of the automaton string. It's the k-th path in the graph
//k is [0..sa_num_substr()-1]
string sa_k_substr(int k) {
    int p = 0;
    char prev = '$';
    string ans = "";
    if(k > sa_num_substr()) return ans; //not exists that k-th string, error
    while(k > 0) {
        prev = '$';
        for(auto el : t[p].next) {
            prev = el.fi;
            if(dp_num_substr[el.se] >= k) break;
            k -= dp_num_substr[el.se];
        }
        if(prev == '$') break; //error
        ans += prev;
        p = t[p].next[prev];
        k--;
    }
    return ans;
}

//lexicographically smallest cyclic shift of the string s
string sa_small_cyclic_shift(string s) {
    int p = 0, cnt = s.length();
    string ans = "";
    sa_ini(s + s); //initialize sa with s+s, the ans is greedy the first path with length s.length()
    while(cnt--) {
        auto el = *(t[p].next.begin()); //take greedy the first edge
        ans += el.fi;
        p = el.se;
    }

    return ans;
}
//int sa_num_ocurrences(string w); //Better use Aho-Corasick
//Test of the automaton string, the number of the substrings and the substrings, sorted
void sa_test1() {
    ll i, n;
    sa_ini("test");
    n = sa_num_substr();
    cout << n << endl;
    for(i = 0; i < n; i++)
        cout << sa_k_substr(i) << endl;
}
```