Echo's notebook

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

@g++ A.cpp $(FLAGS) -DJUNCO_DEBUG && ./a.out < z.in
```

```
// Iterate over all submasks of a mask. CONSIDER SUBMASK = 0 APART.
for(submask = mask; submask > 0; submask = (submask-1)&mask) {}
```

DP

LCS

```
int LCS() { // Longest Common Subsequence.
    int ns = s.length(), nt = t.length(), i, j;
    vector<vi> dp(ns + 1, vi(nt + 1, 0)); // One empty row and column, dp is 1-index
    for(i = 1; i <= ns; i++) {
        for(j = 1; j <= nt; j++) {
            if(s[i-1] == t[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[ns][nt]; // Length.
}
```

LIS

```
vll v_LIS(vll &v) {
    int i, j, n = v.size();
    vll lis, lis_time(n), ans;
    if(!n) return ans;
    lis.pb(v[0]); lis_time[0] = 1;
    for(i = 1; i < n; i++) {
        if(v[i] > lis.back()) {lis.pb(v[i]); lis_time[i] = lis.size(); continue;}
        int pos = upper_bound(lis.begin(), lis.end(), v[i]) - lis.begin();
        // if(pos > 0 && lis[pos-1] == v[i]) continue; // USE IF YOU WANT STRICTLY
        ↪  INCREASING.
        lis[pos] = v[i];
        lis_time[i] = pos+1;
    }
    j = lis.size();
    for(i = n-1; i >= 0; i--) {
        if(lis_time[i] == j && (ans.empty() || v[i] <= ans.back())) {ans.pb(v[i]); j--;} //
        ↪  <= or <.
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

IO

```
ios::sync_with_stdio(false); cin.tie(nullptr); cout.tie(nullptr);

stringstream ss;
ss << "Hello world";
ss.str("Hello world");
while(ss >> s) cout << s << endl;
ss.clear();
```

Geometry

```cpp
template<typename T>
class Point {
    public:
    static const int LEFT_TURN = 1;
    static const int RIGHT_TURN = -1;
    T x = 0, y = 0;
    Point() = default;
    Point(T _x, T _y) {
        x = _x;
        y = _y;
    }
    friend ostream &operator << (ostream &os, Point<T> &p) {
        os << "(" << p.x << " " << p.y << ")";
        return os;
    }
    bool operator == (const Point<T> other) const {
        return x == other.x && y == other.y;
    }
    // Get the (1º) bottom (2º) left point.
    bool operator < (const Point<T> other) const {
        if(y != other.y) return y < other.y;
        return x < other.x;
    }
    T euclidean_distance(Point<T> other) {
        T dx = x - other.x;
        T dy = y - other.y;
        return sqrt(dx*dx + dy*dy);
    }
    T euclidean_distance_squared(Point<T> other) {
        T dx = x - other.x;
        T dy = y - other.y;
        return dx*dx + dy*dy;
    }
    T manhatan_distance(Point<T> other) {
        return abs(other.x - x) + abs(other.y - y);
    }
    // Get the height of the triangle with base b1, b2.
    T height_triangle(Point<T> b1, Point<T> b2) {
        if(b1 == b2 || *this == b1 || *this == b2) return 0; // It's not a triangle.
        T a = euclidean_distance(b1);
        T b = b1.euclidean_distance(b2);
        T c = euclidean_distance(b2);
        T d = (c*c-b*b-a*a)/(2*b);
        return sqrt(a*a - d*d);
    }
    int get_quadrant() {
        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; // Point (0, 0).
    }
    // Relative quadrant respect the point other, not the origin.
    int get_relative_quadrant(Point<T> other) {
        Point<T> p(other.x - x, other.y - y);
        return p.get_quadrant();
    }


```

```
61      // Orientation of points *this -> a -> b.
62      int get_orientation(Point<T> a, Point<T> b) {
63          T prod = (a.x - x)*(b.y - a.y) - (a.y - y)*(b.x - a.x);
64          if(prod == 0) return 0;
65          return prod > 0? LEFT_TURN : RIGHT_TURN;
66      }
67      // True if a have less angle than b, if *this->a->b is a left turn.
68      bool angle_cmp(Point<T> a, Point<T> b) {
69          if(get_relative_quadrant(a) != get_relative_quadrant(b))
70              return get_relative_quadrant(a) < get_relative_quadrant(b);
71          int ori = get_orientation(a, b);
72          if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
73          return ori == LEFT_TURN;
74      }
75      // Anticlockwise sort starting at 1^o quadrant, respect to *this point.
76      void polar_sort(vector<Point<T>> &v) {
77          sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
78      }
79      // Convert v to its convex hull, Do a Graham Scan. O(n log n).
80      void convert_convex_hull(vector<Point<T>> &v) {
81          if(v.size() < 3) return;
82          Point<T> bottom_left = v[0], p2;
83          for(auto p : v) bottom_left = min(bottom_left, p);
84          bottom_left.polar_sort(v);
85          vector<Point<T>> v_input = v; v.clear();
86          for(auto p : v_input) {
87              while(v.size() >= 2) {
88                  p2 = v.back(); v.pop_back();
89                  if(v.back().get_orientation(p2, p) == LEFT_TURN) {
90                      v.pb(p2);
91                      break;
92                  }
93              }
94              v.pb(p);
95          }
96      }
97  };
```

Graphs

UFDS

```cpp
void initialize(int n) {
    for (int i = 0; i < n; ++i) {
        rankk[i] = 0;
        parent[i] = i;
    }
}
int find(int x) {
    if (parent[x] == x)
        return x;
    else
        return parent[x] = find(parent[x]);
}

void Union(int a, int b) {
    int pa = find(a);
    int pb = find(b);
    if (pa == pb) {
        return;
    }
    if (rankk[pa] > rankk[pb]) {
        parent[pb] = pa;
    } else if (rankk[pa] < rankk[pb]) {
        parent[pa] = pb;
    } else {
        parent[pa] = pb;
        rankk[pb]++;
    }
}
```

Articulation points and bridges

```cpp
vector<vi> adyList; // Graph
vi num, low;        // num and low for DFS
int cnt;            // Counter for DFS
int root, rchild;   // Root and number of (DFS) children
vi artic;           // Contains the articulation points at the end
set<pii> bridges;   // Contains the bridges at the end

void dfs(int nparent, int nnode) {
    num[nnode] = low[nnode] = cnt++;
    rchild += (nparent == root);

    for (auto a : adyList[nnode]) {
        if (num[a] == -1) { // Tree edge
            dfs(nnode, a);
            low[nnode] = min(low[nnode], low[a]);
            if (low[a] >= num[nnode]) {
                artic[nnode] = true;
            }
            if (low[a] > num[nnode]) {
                bridges.insert((nnode < a) ? mp(nnode, a) : mp(a, nnode));
            }
        } else if (a != nparent) { // Back edge
            low[nnode] = min(low[nnode], num[a]);
        }
    }
}
void findArticulations(int n) {
    cnt = 0;
    low = num = vi(n, -1);
    artic = vi(n, 0);
```

```
    bridges.clear();

    for (int i = 0; i < n; ++i) {
        if (num[i] != -1) {
            continue;
        }
        root = i;
        rchild = 0;
        dfs(-1, i);
        artic[root] = rchild > 1; //Special case
    }
}
```

Bellman Ford's

```
for(i = 0; i < n - 1; i++) { // Iterate n - 1 times.
    for(auto e : edge) {
        if(dist[e.fi.fi] != inf)
            dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
    }
}
```

Floyd cycle detection

```
void floyd_detection() {
    ll pslow = f(F_0), pfast = f(f(F_0)), iteration = 0;
    while(pslow != pfast) pslow = f(pslow), pfast = f(f(pfast));
    pslow = F_0;
    while(pslow != pfast) pslow = f(pslow), pfast = f(pfast), iteration++;
    cout << "In " << iteration << " coincide with value: " << pslow << endl;
    pfast = f(pfast), iteration++;
    while(pslow != pfast) pfast = f(pfast), iteration++;
    cout << "In " << iteration << " coincide with value: " << pfast << endl;
}
```

Max Flow: Edmond Karp's

```
vector<vector<ll>> adjList;
vector<vector<ll>> adjMat;

void initialize(int n) {
    adjList = decltype(adjList)(n);
    adjMat = decltype(adjMat)(n, vector<ll>(n, 0));
}

map<int, int> p;
bool bfs(int source, int sink) {
    queue<int> q;
    vi visited(adjList.size(), 0);
    q.push(source);
    visited[source] = 1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (u == sink)
            return true;
        for (auto v : adjList[u]) {
            if (adjMat[u][v] > 0 && !visited[v]) {
                visited[v] = true;
                q.push(v);
                p[v] = u;
            }
        }
    }
    return false;
```

```
}
int max_flow(int source, int sink) {
    ll max_flow = 0;
    while (bfs(source, sink)) {
        ll flow = inf;
        for (int v = sink; v != source; v = p[v]) {
            flow = min(flow, adjMat[p[v]][v]);
        }
        for (int v = sink; v != source; v = p[v]) {
            adjMat[p[v]][v] -= flow; // Decrease capacity forward edge
            adjMat[v][p[v]] += flow; // Increase capacity backward edge
        }
        max_flow += flow;
    }
    return max_flow;
}
void addedgeUni(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjMat[orig][dest] = flow;
    adjList[dest].pb(orig); //Add edge for residual flow
}
void addEdgeBi(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjList[dest].pb(orig);
    adjMat[orig][dest] = flow;
    adjMat[dest][orig] = flow;
}
```

Max Flow: Dinic's

```
// O(V^2*E) max flow algorithm. For bipartite matching O(sqrt(V)*E), always faster than
↪  Edmond-Karp.
// Creates layer's graph with a BFS and then it tries all possibles DFS, branching while
↪  the path doesn't reach the sink
struct EdgeFlow {
    ll u, v;
    ll cap, flow = 0; //capacity and current flow
    EdgeFlow(ll _u, ll _v, ll _cap) : u(_u), v(_v), cap(_cap) { }
};

struct Dinic {
    vector<EdgeFlow> edge; //keep the edges
    vector<vll> graph; //graph[u] is the list of their edges
    ll n, n_edges = 0;
    ll source, sink, inf_flow = inf;
    vll lvl; //lvl of the node to the source
    vll ptr; //ptr[u] is the next edge you have to take in order to branch the DFS
    queue<ll> q;

    Dinic(ll _n, ll _source, ll _sink) : n(_n), source(_source), sink(_sink) { //n nodes
        graph.assign(_n, vll());
    }

    void add_edge(ll u, ll v, ll flow) { //u->v with cost x
        EdgeFlow uv(u, v, flow), vu(v, u, 0);
        edge.pb(uv);
        edge.pb(vu);
        graph[u].pb(n_edges);
        graph[v].pb(n_edges+1);
        n_edges += 2;
    }

    bool BFS() {
```

```
        ll u;
        while(q.empty() == false) {
            u = q.front(); q.pop();
            for(auto el : graph[u]) {
                if(lvl[edge[el].v] != -1) {
                    continue;
                }
                if(edge[el].cap - edge[el].flow <= 0) {
                    continue;
                }
                lvl[edge[el].v] = lvl[edge[el].u] + 1;
                q.push(edge[el].v);
            }
        }

        return lvl[sink] != -1;
    }

    ll dfs(ll u, ll min_flow) {
        if(u == sink) return min_flow;
        ll pushed, el;
        for(;ptr[u] < (int)graph[u].size(); ptr[u]++) { //if you can pick ok, else you crop
        ↪  that edge for the current bfs layer
            el = graph[u][ptr[u]];
            if(lvl[edge[el].v] != lvl[edge[el].u] + 1 || edge[el].cap - edge[el].flow <= 0)
            ↪  {
                continue;
            }
            pushed = dfs(edge[el].v, min(min_flow, edge[el].cap - edge[el].flow));
            if(pushed > 0) {
                edge[el].flow += pushed;
                edge[el^1].flow -= pushed;
                return pushed;
            }

        }
        return 0;
    }

    ll max_flow() {
        ll flow = 0, pushed;
        while(true) {
            lvl.assign(n, -1);
            lvl[source] = 0;
            q.push(source);
            if(!BFS()) {
                break;
            }

            ptr.assign(n, 0);
            while(true) {
                pushed = dfs(source, inf_flow);
                if(!pushed) break;
                flow += pushed;
            }
        }
        return flow;
    }
};
```

Hungarian Algorithm

```cpp
const int MAX_N1 = 1002; //number of workers
const int MAX_N2 = 2002; //number of items
int cost[MAX_N1][MAX_N2]; //cost matrix, entries >= 0
int u[MAX_N1+1], v[MAX_N2+1]; //potentials, always cost[i][j] >= u[i] + v[j]
int slack[MAX_N2+1]; //cost[i][j] - u[i] - v[j], always >= 0
int prevy[MAX_N2+1]; //edges of the current path: prev[j0] -> yx[prev[j0]] -> j0. Dont need to reset
bool used[MAX_N2+1]; //visited array
int yx[MAX_N2+1]; //match of the j column
int n1=, n2=, INF = INT_MAX-1; //actual size of workers and items.

//http://e-maxx.ru/algo/assignment_hungary
//Solves MINIMUM Assignment. For maximum change cost[i][j] to Max_entry - cost[i][j] and resize the answer.
//There are 1..n1 rows and 1..n2 columns, ALWAYS n1 <= n2. Complexity(n1 * n1*n2)
//The function use 1-index for variables because it creates a virtual vertex 0
int Hungarian() {
    int i, i0, j, j0, min_j, delta, ans;
    fill(u, u+n1+1, 0);
    fill(v, v+n2+1, 0);
    fill(yx, yx+n2+1, 0);
    for(i = 1; i <= n1; i++) { //Add row by row to the current matching
        yx[0] = i; //connect 0 of set 2 with vertex i
        j0 = 0; //i0 and j0 are the current selected row and column, i and j are just iterators
        fill(slack, slack+n2 + 1, INF);
        fill(used, used + n2 + 1, false);
        do { //while the alternating path is not augmenting path
            used[j0] = true;
            delta = INF;
            i0 = yx[j0];
            for(j = 1; j <= n2; j++) { //get the delta among all columns not used
                if(!used[j]) {
                    int cur = cost[i0-1][j-1] - u[i0] - v[j];
                    if(cur < slack[j]) {
                        slack[j] = cur, prevy[j] = j0;
                    }
                    if(slack[j] < delta) { //try if delta  == 0 break
                        delta = slack[j], min_j = j;
                    }
                }
            }
            for(j = 0; j <= n2; j++) { //add delta in set 1, subtract delta in set 2
                if(used[j]) u[yx[j]] += delta, v[j] -= delta;
                else slack[j] -= delta;
            }
            j0 = min_j;
        } while(yx[j0] != 0);
        do{ //invert the augmenting path
            yx[j0] = yx[prevy[j0]];
            j0 = prevy[j0];
        } while(j0);
    }
    ans = 0;
    for(j = 1; j <= n2; j++) { //recover solution. The matched edges are yx[j]-1 -> j-1
        if(yx[j])
            ans += cost[yx[j]-1][j-1];
    }
    return ans;
}

```

```cpp
// THE ANS IS n1*M_factor - Hungarian().
int M_factor; // Change problem finding the minimum cost to maximum cost, that can be solved by Hungarian
void min_to_max() { //min in cost[i][j] = max in M - cost[i][j].
    int i, j;
    M_factor = 0;
    for(i = 0; i < n1; i++) {
        M_factor = max(M_factor, *max_element(cost[i], cost[i]+n2));
    }
    for(i = 0; i < n1; i++) {
        for(j = 0; j < n2; j++) {
            cost[i][j] = M_factor - cost[i][j];
        }
    }
}
```

Floyd - Warshall

```cpp
// Try to actualize k times the minimum distance.
for(k = 0; k < n; ++k) {
    for(i = 0; i < n; ++i) {
        for(j = 0; j < n; ++j) {
            if(dist[i][k] != INT_MAX && dist[k][j] != INT_MAX)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
       }
    }
}
```

LCA tree

```cpp
const ll LOG_N = 20; //log2(MAX_N) + 4
const ll MAX_N = 1e5 + 3; //1e5, example
vector<vector<ll>> graph2, graph; //graph2 is the bidirectional and graph is the one you ask LCA
//--------------- 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)];
}

//call dfs(0, -1) to root a tree at 0. the graph had to be bidirectional
vector<bool> visitedd;
void dfs(int x, int p) {
    if(visitedd[x]) return;
    visitedd[x] = true;
    if(p != -1) graph[p].pb(x);
    for(auto el : graph2[x]) {
        if(el == p) continue;
        dfs(el, x);
    }
}
```

Kosaraju

```
vector<vi> adyList;  // Graph
vector<int> visited; // Visited for DFS
vector<vi> sccs;     // Contains the SCCs at the end

void dfs(int nnode, vector<int> &v, vector<vi> &adyList) {
    if (visited[nnode]) {
        return;
    }
    visited[nnode] = true;
    for (auto a : adyList[nnode]) {
        dfs(a, v, adyList);
    }
    v.push_back(nnode);
}

void Kosaraju(int n) {
    visited = vi(n, 0);
    stack<int> s = stack<int>();
    sccs = vector<vi>();

    vector<int> postorder;
    for (int i = 0; i < n; ++i) {
        dfs(i, postorder, adyList);
    }
    reverse(all(postorder));

    vector<vi> rAdyList = vector<vi>(n, vi());
    for (int i = 0; i < n; ++i) {
        for (auto v : adyList[i]) {
```

```
            rAdyList[v].push_back(i);
        }
    }

    visited = vi(n, 0);
    vi data;
    for (auto a : postorder) {
        if (!visited[a]) {
            data = vi();
            dfs(a, data, rAdyList);
            if (!data.empty())
                sccs.pb(data);
        }
    }
}
```

Mathematics

Binary operations

```
ll elevate(ll a, ll b) { // b >= 0.
    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) + mod)%mod, mod - 2);
}

ll mul(ll a, ll b) {
    ll ans = 0, neg = (a < 0) ^ (b < 0);
    a = abs(a); b = abs(b);
    while(b) {
        if(b & 1) ans = (ans + a) % mod;
        b >>= 1;
        a = (a + a) % mod;
    }
    if(neg) return -ans;
    return ans;
}
```

Catalan numbers

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Combinatoric numbers

```
 const int MAX_C = 1+66; // 66 is the for long long, C(66, x)
ll Comb[MAX_C][MAX_C];

void calc() {
    int i, j;
    for(i = 0; i < MAX_C; i++) {
        Comb[i][0] = 1;
        Comb[0][i] = 1;
    }
```

```
    for(i = 1; i < MAX_C; i++) {
        for(j = 1; j < MAX_C; j++) {
            if(i+j >= MAX_C) continue;
            Comb[i][j] = Comb[i-1][j] + Comb[i][j-1];
        }
    }
}

ll C(ll i, ll j) {
    return Comb[i-j][j];
}
```

Chinese Remainder

```
const ll MAX = 10;
ll a[MAX], p[MAX], n;
// Given n x == a[i] mod p[i], find x, or -1 if it doesn't exist.
// Let q[i] = (\prod_{i=0}^{n-1} p[j])/p[i].
// x will be = \sum_{i=0}^{n-1} a[i]*q[i]*inv(q[i], mod p[i])
ll chinese_remainder() {
    ll i, j, g, ans = 0, inv1, inv2;
    mod = 1;
    for(i = 0; i < n; i++) { // If the p[i] are not coprimes, do them coprimes.
        a[i] %= p[i]; a[i] += p[i]; a[i] %= p[i];
        for(j = 0; j < i; j++) {
            g = __gcd(p[i], p[j]);
            if((a[i]%g + g)%g != (a[j]%g + g)%g) return -1;
            // Delete the repeated factor at the correct side.
            if (__gcd(p[i]/g, p[j]) == 1) {p[i] /= g; a[i] %= p[i];}
            else {p[j] /= g; a[j] %= p[j];}
        }
    }
    // If you have a supermod, take P = min(P, supermod);
    for(i = 0; i < n; i++) {
        mod *= p[i];
    }
    for(i = 0; i < n; i++) {
        gcdEx(mod/p[i], p[i], &inv1, &inv2);
        ans += mul(a[i], mul(mod/p[i], inv1));
        ans %= mod;
    }
    return (ans%mod + mod) % mod;
}
```

Fast Fourier Transform FFT

```
typedef complex<double> cd;
typedef vector<cd> vcd;

void show(vcd &e) { //for debug
    int cont = 0; for(auto el : e) {cout << " +" << (el.real() > eps ? el.real() : 0) <<
    ↪  "x^" << cont++;} cout << endl;
}
void convolution(vcd &a) { //insert a_i and get y_i = sum_j(a_j*w_i^j)
    int i, n = a.size(); //n power of 2
    if(n == 1) return;
    vcd a_even, a_odd;
    for(i = 0; i < n; i++) { //divide part of FFT
        if(i%2) a_odd.pb(a[i]);
        else a_even.pb(a[i]);
    }
    convolution(a_even); //recursive part
    convolution(a_odd);
    cd wn = polar(1.0, 2*(double)PI/n), w = 1.0; //wn^i are the n roots of n-unity
```

```
    //cd w;
    for(i = 0; i < n/2; i++) {
        //w = polar(1.0, i*2*(double)PI/n); //avoid precission error, but slower
        a[i] = a_even[i] + w*a_odd[i]; //A(wn^k) = Aeven(wn/2^k) + wn^k+Aodd(wn/2^k)
        a[i + n/2] = a_even[i] - w*a_odd[i]; //A(wn^k) = Aeven(wn/2^(k-n/2)) -
        ↪ wn^(k-n/2)+Aodd(wn/2^(k-n/2))
        w = w*wn;
    }
}
void deconvolution(vcd &a) { //insert y_i and get a_i = sum_j(y_j*w_i^-j)/n
    for(auto &el : a) el = conj(el); //you can conjugate wn and do a[i]/n o can
    ↪ conj(a[i])/n
    convolution(a); // The coefficients of the polynomial have to be are real
    for(auto &el : a) el /= (double)a.size();
}
// Calculate \sum_{i=0}^{n-1} a[i]*b[n-i].
vcd FFT(vcd &a, vcd &b) { //multiply polynomial a*b
    //vcd a = {1.0, 2.0}, b = {3.0}, c;// a and b examples of polynomials to multiply, real
    ↪ coefficients
    vcd c;
    if(a.size() < b.size()) swap(a, b);
    int i, n = a.size();
    while(n - LSB(n)) n++, a.pb(0.0); //add 0.0's to the next power of two of the next
    ↪ power of two, 3->8
    n++, a.pb(0.0);
    while(n - LSB(n)) n++, a.pb(0.0);
    while((int) b.size() < n) b.pb(0.0); //the grade of a and b equal.
    convolution(a);
    convolution(b); //if you want a*a then delete this 2º call
    for(i = 0; i < n; i++) c.pb(a[i]*b[i]);
    deconvolution(c);
    return c;
}
```

Euclides

```
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;
}
```

Linear Sieve

```
const int MAX_PRIME = 1e6+5;
bool num[MAX_PRIME]; // If num[i] = false => i is prime.
int num_div[MAX_PRIME]; // Number of divisors of i.
int min_div[MAX_PRIME]; // The smallest prime that divide i.
vector<int> prime;

void linear_sieve(){
    int i, j, prime_size = 0;
    min_div[1] = 1;
    for(i = 2; i < MAX_PRIME; ++i){
```

```
        if(num[i] == false) {prime.push_back(i); ++prime_size; num_div[i] = 1; min_div[i] =
        ↪  i;}

        for(j = 0; j < prime_size && i * prime[j] < MAX_PRIME; ++j){
            num[i * prime[j]] = true;
            num_div[i * prime[j]] = num_div[i] + 1;
            min_div[i * prime[j]] = min(min_div[i], prime[j]);
            if(i % prime[j] == 0) break;
        }
    }
}

bool is_prime(ll n) {
    for(auto el : prime) {
        if(n == el) return true;
        if(n%el == 0) return false;
    }
    return true;
}

vll fact, nfact; // The factors of n and their exponent.
void factorize(int n) { // Up to MAX_PRIME*MAX_PRIME.
    ll cont, prev_p;
    fact.clear(); nfact.clear();
    for(auto p : prime) {
        if(n < MAX_PRIME) break;
        if(n%p == 0) {
            fact.pb(p);
            cont = 0;
            while(n%p == 0) n /= p, cont++;
            nfact.pb(cont);
        }
    }
    if(n >= MAX_PRIME) {
        fact.pb(n);
        nfact.pb(1);
        return;
    }
    while(n != 1) { // When n < MAX_PRIME, factorization in almost O(1).
        prev_p = min_div[n];
        cont = 0;
        while(n%prev_p == 0) n /= prev_p, cont++;
        fact.pb(prev_p);
        nfact.pb(cont);
    }
}
```

BIT Fenweick tree

```
template<typename T>
class BIT{
    vector<T> bit;
    int n;
    public:
    BIT(int _n) {
        n = _n;
        bit.assign(n+1, 0);
    }
    BIT(vector<T> v) {
        n = v.size();
        bit.assign(n+1, 0);
        for(int i = 0; i < n; i++) update(i, v[i]);
    }
```

```
    // Point update.
    void update(int i, T dx) {
        for(i++; i < n+1; i += LSB(i)) bit[i] += dx;
    }
    // query [0, r].
    T query(int r) {
        T ans = 0;
        for(r++; r > 0; r -= LSB(r)) ans += bit[r];
        return ans;
    }
    // query [l, r].
    T query(int l, int r) {
        return query(r) - query(l-1);
    }
    // k-th smallest element inserted.
    int k_element(ll k) { // k > 0 (1-indexed).
        int l = 0, r = n+1, mid;
        if(query(0) >= k) return 0;
        while(l + 1 < r) {
            mid = (l + r)/2;
            if(query(mid) >= k) r = mid;
            else l = mid;
        }
        return r;
    }
};
```

Strings

KMP

```cpp
// Knuth-Morris-Pratt. Search the ocurrences of t (pattern to search) in s (the text).
// O(n). It increases j at most n times and decreases at most n times.
void KMP(string &s, string &t) {
    int n = s.length(), m = t.length(), i, j, len = 0;
    // Longest proper prefix that is also suffix.
    // s[0..lps[i]-1] == s[i-lps[i]+1..i].
    vi lps(m, 0);
    for(i = 1; i < m; i++) {
        if(t[i] == t[len]) {
            len++;
            lps[i] = len;
        } else if(len > 0) {
            len = lps[len - 1];
            i--;
        }
    }
    for(i = 0, j = 0; i < n; i++) {
        if(s[i] == t[j]) {
            j++;
            if(j == m) {
                echo("Pattern found at:", i-j+1);
                // You will math at least lps[j-1] chars.
                j = lps[j - 1];
            }
        } else if(j > 0) {
            j = lps[j - 1];
            i--;
        }
    }
}
```

Longes Palindromic Substring

```cpp
// LPS Longest Palindromic Substring, O(n).
void Manacher(string &str) {
    char ch = '#'; // '#' a char not contained in str.
    string s(1, ch), ans;
    for(auto c : str) {s += c; s += ch;}
    int i, n = s.length(), c = 0, r = 0;
    vi lps(n, 0);
    for(i = 1; i < n; i++) {
        // lps[i] >= it's mirror, but falling in the interval [L..R]. L = c - (R - c).
        if(i < r) lps[i] = min(r - i, lps[c - (i - c)]);
        // Try to increase.
        while(i-lps[i]-1 >= 0 && i+lps[i]+1 < n && s[i-lps[i]-1] == s[i+lps[i]+1])
        ↪  lps[i]++;
        // Update the interval [L..R].
        if(i + lps[i] > r) c = i, r = i + lps[i];
    }
    // Get the longest palindrome in ans.
    int pos = max_element(lps.begin(), lps.end()) - lps.begin();
    for(i = pos - lps[pos]; i <= pos + lps[pos]; i++) {
        if(s[i] != ch) ans += s[i];
    }
    //cout << ans.size() << "\n";
}
```

Z-algorithm

```cpp
// Search the ocurrences of t (pattern to search) in s (the text).
// O(n + m). It increases R at most 2n times and decreases at most n times.
// z[i] is the longest string s[i..i+z[i]-1] that is a prefix = s[0..z[i]-1].
void z_algorithm(string &s, string &t) {
    s = t + "$" + s; // "£" is a char not present in s nor t.
    int n = s.length(), m = t.length(), i, L = 0, R = 0;
    vi z(n, 0);
    // s[L..R] = s[0..R-L], [L, R] is the current window.
    for(i = 1; i < n; i++) {
        if(i > R) { // Old window, recalculate.
            L = R = i;
            while(R < n && s[R] == s[R-L]) R++;
            R--;
            z[i] = R - L + 1;
        } else {
            if(z[i-L] < R - i) z[i] = z[i-L]; // z[i] will fall in the window.
            else { // z[i] can fall outside the window, try to increase the window.
                L = i;
                while(R < n && s[R] == s[R-L]) R++;
                R--;
                z[i] = R - L + 1;
            }
        }
        if(z[i] == m) { // Match found.
            //echo("Pattern found at: ", i-m-1);
        }
    }
}
```

Suffix-Automaton

```cpp
#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;
```

```cpp
}
//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;
```

```
}
```

Aho-Corasick

```
//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;
}

```

```cpp
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;
}
```

Ad-hoc

```cpp
int is_leap_year(int y) {
    if(y%4 || (y%100==0 && y%400)) return 0;
    return 1;
}
int days_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int days_month_accumulate[12] = {31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};

// d 1-index, m 1-index.
int date_to_num(int d, int m, int y) {
    m -= 2;
    int sum = d;
    if(m >= 1) sum += is_leap_year(y);
    y--;
    if(m >= 0) sum += days_month_accumulate[m];
    if(y >= 0) {
        sum += 365*y;
        sum += y/4 -y/100 + y/400;
    }
    return sum;
}

int nd, nm, ny; // Tiny optimization, binary search the year, month and day.
void num_to_date(int num) {
    nd = 1; nm = 1; ny = 2020; // The date searched is >= this date.
    while(date_to_num(nd, nm, ny) <= num) ny++;
```

```
    ny--;
    while(nm < 12 && date_to_num(nd, nm, ny) <= num) nm++;
    nm--;
    while(date_to_num(nd, nm, ny) <= num) nd++;
    nd--;
}
```

Hash Set

```
const int MAX = 2*1e5+5;
ll val[MAX]; // For random numbers and not index use f with random xor.

void ini() { // CALL ME ONCE.
    srand(time(0));
    for(int i = 0; i < MAX; i++) val[i] = rand();
}

// Hash_set contains a set of indices [0..MAX-1] with duplicates.
// a[i] = sum_x{val_x} % mod p[i].
class Hash_set {
    public:
    vll p = {1237273, 1806803, 3279209}; // Prime numbers.
    vll a = {0, 0, 0};
    int n = 3; // n = p.size();
    // Insert index x.
    void insert(int x) {
        for(int i = 0; i < n; i++) a[i] = (a[i] + val[x]) % p[i];
    }
    // Insert all the elements of hs.
    void insert (Hash_set hs) {
        for(int i = 0; i < n; i++) a[i] = (a[i] + hs.a[i]) % p[i];
    }

    bool operator == (Hash_set hs) {
        for(int i = 0; i < n; i++) if(a[i] != hs.a[i]) return false;
        return true;
    }
};
```