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) {}
```

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();
```

Dates

```
// Change here and date_to_num.
ll is_leap_year(ll y) {
    // if(y%4 || (y%100==0 && y%400)) return 0; //
    ↪  Complete leap year.
    if(y%4 != 0) return 0; // Restricted leap year.
    return 1;
}
ll days_month[12] = {31, 28, 31, 30, 31, 30, 31,
↪  31, 30, 31, 30, 31};
ll days_month_accumulate[12] = {31, 59, 90, 120,
↪  151, 181, 212, 243, 273, 304, 334, 365};

// d 1-index, m 1-index.
ll date_to_num(ll d, ll m, ll y) {
    ll sum = d;
    m -= 2;
    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; // Complete
        ↪  leap year.
        sum += y/4; // Restricted leap year.
    }
    return sum;
}

// Tiny optimization, binary search the year, month
↪  and day.
void num_to_date(ll num, ll &d, ll &m, ll &y) {
    d = 1; m = 1; y = 0; // The date searched is >=
    ↪  this date.
    while(date_to_num(d, m, y) <= num) y++;
    y--;
    while(date_to_num(d, m, y) <= num) m++;
    m--;
    while(date_to_num(d, m, y) <= num) d++;
    d--;
}
```

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();
    }
    // Orientation of points *this -> a -> b.
    int get_orientation(Point<T> a, Point<T> b) {
        T prod = (a.x - x)*(b.y - a.y) - (a.y - y)*(b.x - a.x);
        if(prod == 0) return 0;
        return prod > 0? LEFT_TURN : RIGHT_TURN;
    }
    // True if a have less angle than b, if *this->a->b is a left turn.
    bool angle_cmp(Point<T> a, Point<T> b) {
        if(get_relative_quadrant(a) != get_relative_quadrant(b))
            return get_relative_quadrant(a) < get_relative_quadrant(b);
        int ori = get_orientation(a, b);
        if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
        return ori == LEFT_TURN;
    }
    // Anticlockwise sort starting at 1º quadrant, respect to *this point.
    void polar_sort(vector<Point<T>> &v) {
        sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
    }
    // Convert v to its convex hull, Do a Graham Scan. O(n log n).
    void convert_convex_hull(vector<Point<T>> &v) {
        if(v.size() < 3) return;
        Point<T> bottom_left = v[0], p2;
        for(auto p : v) bottom_left = min(bottom_left, p);
        bottom_left.polar_sort(v);
        vector<Point<T>> v_input = v; v.clear();
        for(auto p : v_input) {
            while(v.size() >= 2) {
                p2 = v.back(); v.pop_back();
                if(v.back().get_orientation(p2, p) == LEFT_TURN) {
                    v.pb(p2);
                    break;
                }
            }
            v.pb(p);
        }
    }
};
```

Graphs

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
    }
}
```

Max Flow: Edmond Karp's $\mathcal{O}(VE^2)$

```cpp
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;
}
```

Bellman Ford's

```cpp
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

```cpp
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: Dinic's $\mathcal{O}(V^2E)$

```cpp
// 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
// The rows are jobs, the columns are workers
pair<ll, vl> hungarian(vector<vl> &matrix) {
    int n = matrix.size(), m = matrix[0].size();
    vl jobP(n), workerP(m + 1), matched(m + 1, -1);

    vl dist(m + 1, inf);
    vi from(m + 1, -1), seen(m + 1, 0);

    for (int i = 0; i < n; ++i) {
        int cWorker = m;
        matched[cWorker] = i;
        std::fill(all(dist), inf);
        std::fill(all(from), -1);
        std::fill(all(seen), false);

        while (matched[cWorker] != -1) {
            seen[cWorker] = true;
            int i0 = matched[cWorker];
            int nextWorker = -1;
            ll delta = inf;

            for (int worker = 0; worker < m;
            ↪  ++worker) {
                if (seen[worker])
                    continue;
                ll candidateDistance =
                ↪  matrix[i0][worker];
                candidateDistance += -jobP[i0] -
                ↪  workerP[worker];

                if(candidateDistance<dist[worker]){
                    dist[worker] =
                    ↪  candidateDistance;
                    from[worker] = cWorker;
                }
                if (dist[worker] < delta) {
                    delta = dist[worker];
                    nextWorker = worker;
                }
            }
            for (int j = 0; j <= m; ++j) {
                if (seen[j]) {
                    jobP[matched[j]] += delta;
                    workerP[j] -= delta;
                } else {
                    dist[j] -= delta;
                }
            }
            cWorker = nextWorker;
        }
        while (cWorker != m) {
            int prevWorker = from[cWorker];
            matched[cWorker] = matched[prevWorker];
            cWorker = prevWorker;
        }
    }
    ll ans = -workerP[m];
    vl rowMatchesWith(n);
    for (int j = 0; j < m; ++j) {
        if (matched[j] != -1) {
            rowMatchesWith[matched[j]] = j;
        }
    }
    return {ans, std::move(rowMatchesWith)};
}
```

Floyd - Warshall: k->i->j

Kosaraju

```cpp
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);
        }
    }
}
```

LCA tree

```cpp
const int MAX_N = 1e5 + 5;
const int MAX_LOG_N = 18;
int n;
vector<vi> graph; // Directed graph, allways reserve memory for it.
vector<vi> bigraph; // Undirected graph, reserve memory only if needed.

int level[MAX_N]; // level of the node rooted.
int parent[MAX_N][MAX_LOG_N]; // parent[i][j] is the parent 2^j of the node i.

vector<bool> visited_bigraph;
// root_graph(u, -1) roots the bigraph at node u.
void root_graph(int u, int p) {
    if(p == -1) visited_bigraph.assign(n, false);
    for(auto v : bigraph[u]) {
        if(v == p) continue;
        graph[u].pb(v);
        root_graph(v, u);
    }
}

// Calcule the level and parent 1. Don't call.
void dfs_level(int u, int p) {
    parent[u][0] = p;
    level[u] = level[p] + 1;
    for(auto v : graph[u]) {
        if(v == p) continue;
        dfs_level(v, u);
    }
}
// Builds the LCA.
void build_lca(int root) {
    int i, j;
    level[root] = -1;
    dfs_level(root, root); // The parent of the root is itself.
    for(j = 1; j < MAX_LOG_N; j++) {
        for(i = 0; i < MAX_N; i++) {
            parent[i][j] = parent[parent[i][j - 1]][j - 1];
        }
    }
}
// Calculates the LCA(u, v) in O(log n).
int lca(int u, int v) {
    if(level[u] > level[v]) swap(u, v);
    int i, d = level[v] - level[u];
    for(i = MAX_LOG_N - 1; i >= 0; i--) {
        if(is_set(d, i)) v = parent[v][i];
    }
    if(u == v) return u;
    for(i = MAX_LOG_N - 1; i >= 0; i--) {
        if(parent[u][i] != parent[v][i])
            u = parent[u][i], v = parent[v][i];
    }
    return parent[u][0];
}
// Calculates the distance(u, v) in a tree in O(log n).
int dist(int u, int v) {
    return level[u] + level[v] - 2 * level[lca(u, v)];
}
```

Mathematics

Binary operations

```cpp
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

```cpp
 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

```cpp
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;
}
```

Euclides

```cpp
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;
}
```

Hash Set

```cpp
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();

    void insert(int x) {  // Insert index 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;
    }
};
```

Hash of pairs

```cpp
// Use unordered_set<pii, pair_hash> us or unordered_map<pii, int, pair_hash> um;
struct pair_hash
{
    template <class T1, class T2>
    size_t operator () (pair<T1, T2> const &pair) const
    {
        size_t h1 = hash<T1>()(pair.first);
        size_t h2 = hash<T2>()(pair.second);

        return (h1 ^ 0b11001001011001101) + (0b011001010011100111 ^ h2);
    }
};
```

Linear Sieve

```cpp
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;

```

```cpp
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;
        }
    }
}
```

```cpp
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);
    }
}
```

Suffix Array

```cpp
class SuffixArray {
    public:
    int n;
    string s;
    vi p; // p[i] is the position in the order array of the ith suffix (s[i..n-1]).
    vi c; // c[i] is the equivalence class of the ith suffix. When build, c[p[i]] = i, inverse.
    // dont use lcp[0] = 0.
    vi lcp; // lcp[i] is the longest common prefix in s[p[i-1]..n-1] and s[p[i]..n-1].
    // To get lcp(s[i..n-1], s[j..n-1) is min(lcp[c[i]+1], lcp[c[j]]) (use SegTree).
    void radix_sort(vector<pair<pii, int>> &v) { // O(n).
        vector<pair<pii, int>> v2(n);
        vi freq(n, 0); // first frequency and then the index of the next item.
        int i, sum = 0, temp;
        for(i = 0; i < n; i++) freq[v[i].fi.se]++; // Sort by second component.
        for(i = 0; i < n; i++) {temp = freq[i]; freq[i] = sum; sum += temp;}
        for(i = 0; i < n; i++) {v2[freq[v[i].fi.se]] = v[i]; freq[v[i].fi.se]++;}
        freq.assign(n, 0); sum = 0;
        for(i = 0; i < n; i++) freq[v2[i].fi.fi]++; // Sort by first component.
        for(i = 0; i < n; i++) {temp = freq[i]; freq[i] = sum; sum += temp;}
        for(i = 0; i < n; i++) {v[freq[v2[i].fi.fi]] = v2[i]; freq[v2[i].fi.fi]++;}
    }
    SuffixArray() = default;
    SuffixArray(string &_s) {
        s = _s;
        s += "$"; // smaller char to end the string.
        n = s.size();
        int i, k;
        p.assign(n, 0);
        c.assign(n, 0);
        vector<pii> v1(n); // temporal vector to sort.
        for(i = 0; i < n; i++) v1[i] = mp(s[i], i);
        sort(v1.begin(), v1.end());
        for(i = 0; i < n; i++) p[i] = v1[i].se;
        c[p[0]] = 0;
        for(i = 1; i < n; i++) {
            if(v1[i].fi == v1[i - 1].fi) c[p[i]] = c[p[i - 1]];
            else c[p[i]] = c[p[i - 1]] + 1;
        }
        k = 0; // in k+1 iterations sort strings of length 2^(k+1).
        while(c[p[n-1]] != n-1) { // At most ceil(log2(n)).
            vector<pair<pii, int>> v2(n); // temporal vector to sort.
            for(i = 0; i < n; i++) v2[i] = mp(mp(c[i], c[(i + (1 << k)) % n]), i);
```

```
            radix_sort(v2);
            for(i = 0; i < n; i++) p[i] = v2[i].se;
            c[p[0]] = 0;
            for(i = 1; i < n; i++) {
                if(v2[i].fi == v2[i - 1].fi)
                ↪ c[p[i]] = c[p[i - 1]];
                else c[p[i]] = c[p[i - 1]] + 1;
            }
            k++;
        }
    }
    void show_suffixes() { // IMPORTANT use this to
    ↪ debug.
        for(int i = 0; i < n; i++) cout << i << " "
        ↪ << p[i] << " " << s.substr(p[i]) <<
        ↪ endl;
        if(!lcp.empty()) cout << "LCP: " << lcp <<
        ↪ endl;
    }
    // cmp s with t. return -1 if s < t, 1 if s >
    ↪ t, 0 if s == t.
    int cmp_string(int pos, string &t) {
        for(int i = p[pos], j = 0; j < (int)
        ↪ t.size(); i++, j++) {
            if(s[i] < t[j]) return -1; // i < n
            ↪ because s[n-1] = '$'.
            if(s[i] > t[j]) return 1;
        }
        return 0;
    }
    // Count the number of times t appears in s.
    int count_substring(string &t) {
        int l = -1, r = n, mid, L, R;
        while(l + 1 < r) { //
        ↪ -1,...,-1=L,0,...,0,1=R...1.
            mid = (l + r) / 2;
            if(cmp_string(mid, t) < 0) l = mid;
            else r = mid;
        }
        L = l;
        l = -1; r = n;
        while(l + 1 < r) {
            mid = (l + r) / 2;
            if(cmp_string(mid, t) <= 0) l = mid;
            else r = mid;
        }
        R = r;
        return R - L - 1;
    }
    // O(n) build. At most 2n lcp++ and n lcp--;
    void build_lcp() {
        lcp.assign(n, 0);
        for(int i = 0; i < n - 1; i++) {
            if(i > 0) lcp[c[i]] = max(lcp[c[i - 1]]
            ↪ - 1, 0);
            while(s[i + lcp[c[i]]] == s[p[c[i] - 1]
            ↪ + lcp[c[i]]]) lcp[c[i]]++;
        }
    }
    ll number_substrings() {
        ll ans = 0, i;
        for(i = 1; i < n; i++) {
            ans += n - p[i-1] - lcp[i]; // Length
            ↪ of the suffix - lcp with the next
            ↪ suffix.
        }
        ans += n - p[n - 1]; // Plus the last
        ↪ suffix.
        return ans - n; // Remove the '$' symbol on
        ↪ n substrings.
    }
};
string LCS(string s, string &t) {
    int mx = 0, mxi = 0, i, n2 = t.length();
    string ans = "";
    s += "@" + t; // Concatenate with a special
    ↪ char.
    SuffixArray sa(s);
    sa.build_lcp();
    for(i = 1; i < sa.n; i++) {
        // Suffix of s and before suffix of t.
        if(sa.n - sa.p[i] > n2 + 2 && sa.n -
        ↪ sa.p[i-1] <= n2 + 1) {
            if(sa.lcp[i] > mx) mx = sa.lcp[i], mxi
            ↪ = i;
        }
        // Suffix of t and before suffix of s.
        if(sa.n - sa.p[i] <= n2 + 1 && sa.n -
        ↪ sa.p[i-1] > n2 + 2) {
            if(sa.lcp[i] > mx) mx = sa.lcp[i], mxi
            ↪ = i;
        }
    }
    return sa.s.substr(sa.p[mxi], mx);
}
```

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;
    }
    T query(int r) { // query [0, r].
        T ans = 0;
        for(r++; r > 0; r -= LSB(r)) ans += bit[r];
        return ans;
    }
    T query(int l, int r) { // query [l, 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
template <typename T>
vi prefixFun(const T &s, int n) {
    vi res(n);
    for (int i = 1; i < n; ++i) {
        int j = res[i - 1];
        while (j > 0 && s[i] != s[j]) {
            j = res[j - 1];
        }
        res[i] = j + (s[i] == s[j]);
    }
    return res;
}

template <typename T>
int kmpSearch(const T &text, int n,
              const T &pattern, int m,
              const vi &patternPre) {

    int count = 0;
    int j = 0;
    for (int i = 0; i < n; ++i) {
        while (j > 0 && text[i] != pattern[j]) {
            j = max(0, patternPre[j] - 1);
        }
        j += (text[i] == pattern[j]);
        if (j == m) {
            count++;
            j = patternPre[j - 1];
        }
    }
    return count;
}
```

Longest 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;
    int 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 {
            // z[i] will fall in the window.
            if(z[i-L] < R - i) z[i] = z[i-L];
            // z[i] can fall outside the window,
            // try to increase the window.
            else {
                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);
        }
    }
}
```