Z-Algorithm and KMP

```cpp
// z[i] is the length of the longest substring starting at i that is prefix s
vector<int> z_function(const string& s) {
  int n = s.size();
  vector<int> z(n, 0);
  int L = 0, R = 0;
  for (int i = 1; i < n; i++) {
    if (i <= R) z[i] = min(R - i + 1, z[i - L]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if (i + z[i] - 1 > R) {
      L = i;
      R = i + z[i] - 1;
    }
  }
  return z;
}

// find all starting indices in text where pattern occurs
vector<int> z_match(const string& text, const string& pattern) {
  string concat = pattern + "#" + text;
  vector<int> z = z_function(concat);
  vector<int> positions;
  int m = pattern.size();
  for (int i = m + 1; i < (int)concat.size(); i++) {
    if (z[i] == m) positions.push_back(i - m - 1);
  }
  return positions;
}

// p[i] is the length of the longest prefix and suffix of s[0..i]
vector<int> prefix_function(const string& s) {
  int n = s.size();
  vector<int> pi(n, 0);
  for (int i = 1; i < n; i++) {
    int 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;
}

// find all starting indices in text where pattern occurs
vector<int> kmp_match(const string& text, const string& pattern) {
  int n = text.size(), m = pattern.size();
  if (m == 0) return {};
  vector<int> pi = prefix_function(pattern);
  vector<int> positions;
  int j = 0;
  for (int i = 0; i < n; i++) {
    while (j > 0 && text[i] != pattern[j]) j = pi[j - 1];
    if (text[i] == pattern[j]) j++;
    if (j == m) {
      positions.push_back(i - m + 1);
      j = pi[j - 1];
    }
  }
  return positions;
}
```

Rolling Hash

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

int P1 = 31; // P2 = 37; // 131 // 1315423911ULL
int M1 = 1e9 + 7; // M2 = 1e9 + 9; // or no MOD with unsigned long long
const int mxn = 1e6 + 5;
string s; int n;
ll hash_f[mxn]; // forward hashes: hash of substring of length i
ll hash_r[mxn]; // reverse hashes
ll pow_p1[mxn]; // precomuted powers for substring hashes
void precompute() {
  pow_p1[0] = 1;
  for (int i = 1; i < mxn; ++i) pow_p1[i] = (pow_p1[i-1] * P1) % M1;
}
ll compute_hash() {
  // forward hashes: s[0]*p^n + s[1]*p^(n-1) + s[2]*p^(n-2) .. mod m
  hash_f[0] = hash_r[n] = 0;
  for (int i = 0; i < n; ++i) {
    hash_f[i + 1] = ((hash_f[i] * P1)%M1 + (s[i]-'a'+1)) % M1;
  }
  for (int i = n - 1; i >= 0; --i) {
    hash_r[i] = ((hash_r[i + 1] * P1)%M1 + (s[i]-'a'+1)) % M1;
  }
  return hash_f[n]; // full string hash
}
ll change_char(ll h1, int pos, char newc, char oldc) {
  // Each character contributes [char_value * P^(n-pos-1)]
  ll exp = pow_p1[n-pos-1];
  return ((h1 - ((oldc-'a'+1) * exp)%M1 + ((newc-'a'+1) * exp)%M1) + M1) % M1;
}
ll get(int l, int r, bool get_fwd = true) { // 0-based indexing
  ll fwd = ((hash_f[r+1] - (hash_f[l] * pow_p1[r-l+1])%M1) + M1) % M1;
  ll rev = ((hash_r[l] - (hash_r[r+1] * pow_p1[r-l+1])%M1) + M1) % M1;
  return get_fwd ? fwd : rev;
}
bool is_palindrome(int l, int r) { return get(l,r) == get(l,r,0); }
bool has_infix(int len) {
  ll h1 = get(0, len - 1);
  for (int i = 1; i < n - len; ++i) {
    if (h1 == get(i, i+len-1)) return true;
  }
  return false;
}
vector<int> prefix_suffixes() {
  vector<int> res; // prefix lengths that are also suffixes
  for (int i = 0; i < n; ++i) {
    if (get(0, i) == get(n-i-1, n - 1)) {
      res.push_back(i + 1);
    }
  }
  return res;
}
```

Trie

```
struct Trie {
  char base = 'a';
  struct Node {
    int next[26]; bool isEnd; int freq;
    Node() : isEnd(false), freq(0) { fill(next, next + 26, -1); }
  };
  vector<Node> tree;
  Trie() { tree.push_back(Node()); }
  void insert(const string& s) {
    int p = 0;
    tree[p].freq++;
    for (char c : s) {
      int idx = c - base;
      if (tree[p].next[idx] == -1) {
        tree[p].next[idx] = tree.size();
        tree.push_back(Node());
      }
      p = tree[p].next[idx];
      tree[p].freq++;
    }
    tree[p].isEnd = true;
  }
  bool search(const string& s) {
    int p = 0;
    for (char c : s) {
      int idx = c - base;
      if (tree[p].next[idx] == -1) return false;
      p = tree[p].next[idx];
    }
    return tree[p].isEnd;
  }
  bool startsWith(const string& s) {
    int p = 0;
    for (char c : s) {
      int idx = c - base;
      if (tree[p].next[idx] == -1) return false;
      p = tree[p].next[idx];
    }
    return true;
  }
};

// number of ways to form string S using words in the dictionary
int word_combinations(const string& s, const vector<string>& words) {
  const int MOD = 1e9 + 7;
  int n = s.size();
  Trie trie;
  for (auto& word : words) trie.insert(word);
  vector<int> dp(n + 1, 0);
  dp[0] = 1; // base case: empty string
  for (int i = 0; i < n; i++) {
    int p = 0; // start at root of trie
    for (int j = i; j < n; j++) {
      int idx = s[j] - 'a';
      if (trie.tree[p].next[idx] == -1) break; // no matching prefix
      p = trie.tree[p].next[idx];
      if (trie.tree[p].isEnd) dp[j + 1] = (dp[j + 1] + dp[i]) % MOD; // extend ways
    }
  }
  return dp[n];
}
```

Game Theory

```cpp
/*
Normal play convention: player who makes the last move wins
Partisan game: available moves are dependent on a player's current state
Impartial game: both players have the same moves at any turn

Nim-sum of a position of the bitwise xor of all pile sizes.
- A Nim position is losing iff the Nim-sum is zero
- Otherwise it is winning

Theorem:
Every impartial game position is equivalent to a Nim heap of some size g.
- g is the Grundy number (Sprague-Grundy value) of the position.

For a position X, moves(X) is the set of available moves:
grundy(X) = mex{grundy(Y) | Y in moves(X)} -- minimum excluded non-negative int
Base:
- grundy(position with no moves) = 0
- grundy(X) = 0 if X is a losing position for the current turn
- grundy(X) != 0 are winning positions

If a position is made up of several independent subgames:
g(X) = g(Y) xor g(Z) xor ...
*/

int mex(const vector<int>& v) {
  int n = v.size();
  vector<char> used(n + 1, 0);
  for (int x : v)
    if (x >= 0 && x <= n) used[x] = 1;
  for (int i = 0; i <= n; ++i)
    if (!used[i]) return i;
  return n;
}

vector<int> sprague(int N, const vector<int>& moves) {
  vector<int> sg(N + 1, 0); // dp with base [0] = 0
  for (int x = 1; x <= N; ++x) {
    vector<int> nxt;
    for (int mv : moves) {
      if (x - mv >= 0) nxt.push_back(sg[x - mv]);
    }
    // grundy = mex{grundy values}
    sg[x] = mex(nxt);
  }
  return sg;
}

int nim_sum(const vector<int>& v) {
  int x = 0;
  for (int g : v) x ^= g;
  return x;
}
```

Matrix Operations

```
const int N = 2; // 2x2 matrix
const int MOD = 1e9;
struct matrix_t {
  int x[N + 1][N + 1]; // 1-based matrix!
  // zero matrix by default, or identity if v=1
  matrix_t(int v = 0) {
    memset(x, 0, sizeof(x));
    if (v != 0) {
      for (int i = 1; i <= N; ++i) x[i][i] = v % MOD;
    }
  }
  matrix_t operator*(const matrix_t& r) const {
    matrix_t p;
    for (int k = 1; k <= N; ++k) {
      for (int i = 1; i <= N; ++i) {
        if (x[i][k] == 0) continue;
        for (int j = 1; j <= N; ++j) {
          p.x[i][j] = (p.x[i][j] + 1LL * x[i][k] * r.x[k][j]) % MOD;
        }
      }
    }
    return p;
  }
  matrix_t power(ll p) const {
    matrix_t r(1);
    matrix_t a = *this;
    while (p) {
      if (p & 1) r = r * a;
      a = a * a;
      p >>= 1;
    }
    return r;
  }
  // mod must be prime for fermat's little theorem
  static int mod_inv(int a) {
    int res = 1, b = a, e = MOD - 2;
    while (e) {
      if (e & 1) res = 1LL * res * b % MOD;
      b = 1LL * b * b % MOD;
      e >>= 1;
    }
    return res;
  }

  matrix_t inverse() const {
    matrix_t a = *this;
    matrix_t inv(1);

    for (int i = 1; i <= N; ++i) {
      int pivot = -1;
      for (int j = i; j <= N; ++j) {
        if (a.x[j][i] != 0) {
          pivot = j;
          break;
        }
      }
      if (pivot == -1) throw runtime_error("Matrix is singular");
      if (pivot != i) {
        swap(a.x[i], a.x[pivot]);
        swap(inv.x[i], inv.x[pivot]);
      }
```

```
      int inv_pivot = mod_inv(a.x[i][i]);
      for (int j = 1; j <= N; ++j) {
        a.x[i][j] = 1LL * a.x[i][j] * inv_pivot % MOD;
        inv.x[i][j] = 1LL * inv.x[i][j] * inv_pivot % MOD;
      }
      for (int j = 1; j <= N; ++j) {
        if (i == j) continue;
        int factor = a.x[j][i];
        for (int k = 1; k <= N; ++k) {
          a.x[j][k] = (a.x[j][k] - 1LL * factor * a.x[i][k] % MOD + MOD) % MOD;
          inv.x[j][k] =
              (inv.x[j][k] - 1LL * factor * inv.x[i][k] % MOD + MOD) % MOD;
        }
      }
    }
    return inv;
  }
};

// simple 2x2 matrix multiplication
void mul(ll a[2][2], ll b[2][2]) {
  ll res[2][2];
  res[0][0] = (a[0][0] * b[0][0] % MOD + a[0][1] * b[1][0] % MOD) % MOD;
  res[0][1] = (a[0][0] * b[0][1] % MOD + a[0][1] * b[1][1] % MOD) % MOD;
  res[1][0] = (a[1][0] * b[0][0] % MOD + a[1][1] * b[1][0] % MOD) % MOD;
  res[1][1] = (a[1][0] * b[0][1] % MOD + a[1][1] * b[1][1] % MOD) % MOD;
  memcpy(a, res, sizeof(res));
}

int main() {
  int n = 10;
  matrix_t fib; // transformation matrix
  fib.x[1][1] = 1;
  fib.x[1][2] = 1;
  fib.x[2][1] = 1;
  fib.x[2][2] = 0;

  // Claim to our induction over the transformation matrix
  // [0 1]^n [f(0)] = [ f(n) ]
  // [1 1]   [f(1)] = [f(n+1)]
  matrix_t result = fib.power(n);
  cout << "fib(" << n << ") = " << (result.x[1][1] * 0 + result.x[1][2] * 1) % MOD << endl;

  // alternative simpler approach:
  ll mat[2][2] = {{0, 1}, {1, 1}};
  ll res[2][2] = {{1, 0}, {0, 1}}; // identity matrix
  // calculate mod-pow
  while (n) {
    if (n & 1) mul(res, mat);
    mul(mat, mat);
    n >>= 1;
  }
  // mult by [f(0), f(1)] vector and get f(n)
  cout << (res[0][0] * 0 + res[0][1] * 1) << "\n";
}
```

Graph Algorithms

```
const ll INF = 1e18;
const int MAX_N = 1e5 + 10;
const int MAX_LOG = 20;
int n;
vector<pi> adj_w[MAX_N]; // weighted: {neighbor, weight}
vector<int> adj[MAX_N];  // unweighted

// tree algorithms
int up[MAX_N][MAX_LOG];
int depth[MAX_N];
bool visited[MAX_N];

int timer = 0;
int previsit[MAX_N];
int postvisit[MAX_N];

vector<int> bfs(int start) {
  vector<int> dist(n + 1, -1);
  queue<int> q;
  dist[start] = 0;
  q.push(start);
  while (!q.empty()) {
    int a = q.front();
    q.pop();
    for (int b : adj[a]) {
      if (dist[b] == -1) {
        dist[b] = dist[a] + 1;
        q.push(b);
      }
    }
  }
  return dist;
}

void bfs_dirs() {
  vector<pi> dirs({{0, 1}, {1, 0}, {0, -1}, {-1, 0}});
  queue<pi> q;
  q.push({0, 0});
  while (!q.empty()) {
    pi pos = q.front();
    q.pop();
    for (pi dir : dirs) {
      int ni = pos.first + dir.first;
      int nj = pos.second + dir.second;
    }
  }
}

void dfs(int a) {
  visited[a] = true;
  previsit[a] = timer++;
  for (int b : adj[a]) {
    if (!visited[b]) {
      depth[b] = depth[a] + 1;
      // for undirected, check that (b != up[a][0])
      up[b][0] = a;
      dfs(b);
    }
  }
  postvisit[a] = timer++;
}
```

```cpp
vector<ll> dijkstra(int start) {
  vector<ll> dist(n + 1, INF);
  priority_queue<pi, vector<pi>, greater<pi>> pq; // min heap
  dist[start] = 0;
  pq.push({0, start});
  while (!pq.empty()) {
    auto [d, a] = pq.top();
    pq.pop();
    if (d > dist[a]) continue;
    for (auto [b, w] : adj_w[a]) {
      if (dist[a] + w < dist[b]) {
        dist[b] = dist[a] + w;
        pq.push({dist[b], b});
      }
    }
  }
  return dist;
}

vector<ll> bellman_ford(int start) {
  vector<ll> dist(n + 1, INF);
  dist[start] = 0;
  for (int i = 1; i < n; i++) {
    for (int a = 1; a <= n; a++) {
      if (dist[a] == INF) continue;
      for (auto [b, w] : adj_w[a]) {
        if (dist[b] > dist[a] + w) {
          dist[b] = dist[a] + w;
        }
      }
    }
  }
  for (int a = 1; a <= n; a++) { // check for negative cycles
    if (dist[a] == INF) continue;
    for (auto [b, w] : adj_w[a]) {
      if (dist[b] > dist[a] + w) {
        dist[b] = -INF; // mark as affected by negative cycle
      }
    }
  }
  return dist;
}

vector<vector<ll>> floyd_warshall() {
  vector<vector<ll>> dist(n + 1, vector<ll>(n + 1, INF));
  for (int i = 1; i <= n; i++) {
    dist[i][i] = 0; // self distances
    for (auto [b, w] : adj_w[i]) {
      dist[i][b] = min(dist[i][b], (ll)w); // direct edges
    }
  }
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        if (dist[i][k] < INF && dist[k][j] < INF) {
          dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
      }
    }
  }
  return dist;
}
```

```cpp
vector<int> topological_sort() {
  vector<int> in_degree(n + 1, 0);
  for (int a = 1; a <= n; a++) {
    for (int b : adj[a]) {
      in_degree[b]++;
    }
  }

  queue<int> q;
  for (int a = 1; a <= n; a++) {
    if (in_degree[a] == 0) q.push(a);
  }

  vector<int> topo;
  while (!q.empty()) {
    int a = q.front();
    q.pop();
    topo.push_back(a);
    for (int b : adj[a]) {
      in_degree[b]--;
      if (in_degree[b] == 0) q.push(b);
    }
  }

  if (topo.size() != n) return {}; // cycle exists
  return topo;
}

// implies that there are no odd-length cycles
bool is_bipartite() {
  vector<int> group(n, -1);
  for (int i = 0; i < n; ++i) {
    if (group[i] != -1) continue;
    // bfs from this cc
    queue<int> q;
    q.push(i);
    while (!q.empty()) {
      int a = q.front();
      q.pop();
      for (int b : adj[a]) {
        if (group[b] == -1) {
          group[b] = 1 - group[a];
          q.push(b);
        } else if (group[b] == group[a]) {
          return false;
        }
      }
    }
  }
  return true;
}
```

```cpp
struct LCA {
  int n, MAX_LOG;
  vector<vector<int>> adj; // adjacency list of the tree
  vector<vector<int>> up;  // up[v][j] = 2^j-th ancestor of v
  vector<int> depth;
  LCA(int _n) : n(_n) {
    MAX_LOG = 1;
    while ((1 << MAX_LOG) <= n) ++MAX_LOG;
    adj.assign(n + 1, vector<int>());
    up.assign(n + 1, vector<int>(MAX_LOG, 0));
    depth.assign(n + 1, 0);
  }
  void add_edge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
  void dfs(int v, int p = 0) { // needed for depths
    up[v][0] = p;
    for (int u : adj[v]) {
      if (u != p) {
        depth[u] = depth[v] + 1;
        dfs(u, v);
      }
    }
  }
  void binary_lift() {
    for (int j = 1; j < MAX_LOG; j++) {
      for (int i = 1; i <= n; i++) {
        if (up[i][j - 1] != 0) up[i][j] = up[up[i][j - 1]][j - 1];
      }
    }
  }
  void build(int root = 1) {
    depth[root] = 0;
    dfs(root);
    binary_lift();
  }
  int kth_ancestor(int a, int k) {
    for (int i = 0; i < MAX_LOG; i++) {
      if ((1 << i) & k) {
        a = up[a][i];
        if (a == 0) break; // no ancestor exists
      }
    }
    return a;
  }
  int get_lca(int a, int b) {
    if (depth[a] < depth[b]) swap(a, b);
    int diff = depth[a] - depth[b];
    // lift a to same depth as b
    a = kth_ancestor(a, diff);
    if (a == b) return a;
    // lift both until their parents are equal
    for (int i = MAX_LOG - 1; i >= 0; i--) {
      if (up[a][i] != 0 && up[a][i] != up[b][i]) {
        a = up[a][i];
        b = up[b][i];
      }
    }
    return up[a][0];
  }
  int tree_distance(int a, int b) {
    int lca = get_lca(a, b);
    return depth[a] + depth[b] - 2 * depth[lca];
  }
};
```

Dynamic Programming

```cpp
// dp[w] = max value from using the first i items with total weight <= w
int knapsack_no_rep(vector<int>& weights, vector<int>& val, int CAP) {
  int n = weights.size();
  vector<int> dp(CAP + 1, 0);
  for (int i = 0; i < n; ++i)
    for (int w = CAP; w >= weights[i]; --w)
      dp[w] = max(dp[w], dp[w - weights[i]] + val[i]);
  return dp[CAP];
}
// dp[w] = max value with total weight <= w with unlimited copies of each item
int knapsack_rep(vector<int>& weights, vector<int>& val, int CAP) {
  int n = weights.size();
  vector<int> dp(CAP + 1, 0);
  for (int w = 0; w <= CAP; ++w)
    for (int i = 0; i < n; ++i)
      if (weights[i] <= w) dp[w] = max(dp[w], dp[w - weights[i]] + val[i]);
  return dp[CAP];
}
// inc[i] = length of the longest increasing subsequence ending at index i
int LIS(vector<int>& a, int n) {
  vector<int> inc(n);
  for (int i = 0; i < n; ++i) inc[i] = 1;
  int ans = 0;
  for (int i = 1; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
      if (a[j] < a[i]) {
        inc[i] = max(inc[i], inc[j] + 1);
        ans = max(ans, inc[i]);
      }
  return ans;
}
// min_curr = minimum sum of a subarray ending at current position
int min_subarray_sum(vector<int>& nums) {
  int ans = INT_MAX, min_curr = 0, DEFAULT = 0;
  for (auto& i : nums) {
    min_curr += i;
    ans = min(ans, min_curr);
    min_curr = min(min_curr, DEFAULT);
  }
  return ans;
}
// max_curr = maximum sum of a subarray ending at current position
int max_subarray_sum(vector<int>& nums) {
  int ans = INT_MIN, max_curr = 0, DEFAULT = 0;
  for (auto& i : nums) {
    max_curr += i;
    ans = max(ans, max_curr);
    max_curr = max(max_curr, DEFAULT);
  }
  return ans;
}

// SUBSET SUM OPTIMIZED (result in dp[x])
int n, x; cin >> n >> x;
bitset<mn + 1> dp; // static const int mn
dp[0] = 1;
for (int i = 0; i < n; ++i) {
  int v;
  cin >> v;
  dp |= (dp << v);
}
```

Geometry

```
typedef __int128_t i128;
struct Point {
  ll x, y;
  Point() : x(0), y(0) {}
  Point(ll _x, ll _y) : x(_x), y(_y) {}
  Point operator+(const Point& o) const { return
Point(x + o.x, y + o.y); }
  Point operator-(const Point& o) const { return
Point(x - o.x, y - o.y); }
  Point operator*(ll k) const { return Point(x *
k, y * k); }
  bool operator<(const Point& o) const { return
x != o.x ? x < o.x : y < o.y; }
  bool operator==(const Point& o) const { return x
== o.x && y == o.y; }
};

// from origin, area of parallelogram from vectors
a and b
static inline i128 cross(const Point& a, const
Point& b) {
  return (i128)a.x * b.y - (i128)a.y * b.x; // pos
is counterclockwise a to b
}
// from point o acting as origin, area of
parallelogram from vectors a and b
static inline i128 cross(const Point& o, const
Point& a, const Point& b) {
  return cross(a - o, b - o); // positive is
counterclockwise from a to b
}
static inline i128 dot(const Point& a, const
Point& b) {
  return (i128)a.x * b.x + (i128)a.y * b.y;
}
// distance squared
static inline ll dist2(const Point& a, const
Point& b) {
  ll dx = a.x - b.x;
  ll dy = a.y - b.y;
  return dx * dx + dy * dy;
}
// orient: +1 left turn, -1 right turn, 0
collinear
static inline int orient(const Point& a, const
Point& b, const Point& c) {
  i128 v = cross(a, b, c);
  // from a acting as the origin, we check the
relationship of vectors b and c
  if (v == 0) return 0;
  return (v > 0) ? 1 : -1;
}
// segment containment: check if p lies on segment
ab (inclusive)
bool on_segment(const Point& a, const Point& b,
const Point& p) {
  if (orient(a, b, p) != 0) return false;
  return min(a.x, b.x) <= p.x && p.x <= max(a.x,
b.x) && min(a.y, b.y) <= p.y &&
         p.y <= max(a.y, b.y);
}
```

```
// segment intersection (including endpoints /
collinear)
bool segments_intersect(const Point& a1, const
Point& a2, const Point& b1, const Point& b2) {
  int o1 = orient(a1, a2, b1);
  int o2 = orient(a1, a2, b2);
  int o3 = orient(b1, b2, a1);
  int o4 = orient(b1, b2, a2);
  if (o1 != o2 && o3 != o4) return true;
  if (o1 == 0 && on_segment(a1, a2, b1))
    return true;
  if (o2 == 0 && on_segment(a1, a2, b2))
    return true;
  if (o3 == 0 && on_segment(b1, b2, a1))
    return true;
  if (o4 == 0 && on_segment(b1, b2, a2))
    return true;
  return false;
}

vector<Point> convex_hull(vector<Point> pts, bool
keep_collinear = false) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()),
pts.end());
  int n = pts.size();
  if (n <= 1) return pts;
  vector<Point> lower, upper;
  // lower hull
  for (int i = 0; i < n; ++i) {
    while (lower.size() >= 2) {
      i128 v = cross(lower[lower.size() - 2],
          lower[lower.size() - 1], pts[i]);
      if (v > 0 || (keep_collinear && v == 0)) {
        break;
      }
      lower.pop_back();
    }
    lower.push_back(pts[i]);
  }
  // upper hull
  for (int i = n - 1; i >= 0; --i) {
    while (upper.size() >= 2) {
      i128 v = cross(upper[upper.size() - 2],
          upper[upper.size() - 1], pts[i]);
      if (v > 0 || (keep_collinear && v == 0)) {
        break;
      }
      upper.pop_back();
    }
    upper.push_back(pts[i]);
  }
  // remove duplicates in hull
  lower.pop_back();
  upper.pop_back();
  lower.insert(lower.end(), upper.begin(),
upper.end());
  return lower;
}
```

Data Structures

```
struct DSU {
  vector<int> parent, size;
  DSU(int n) {
    parent.resize(n + 1);
    size.resize(n + 1, 1);
    iota(parent.begin(), parent.end(), 0);
  }
  int find(int a) {
    if (parent[a] == a) return a;
    return parent[a] = find(parent[a]);
  }
  bool same(int a, int b) {
    return find(a) == find(b);
  }
  void merge(int a, int b) {
    a = find(a);
    b = find(b);
    if (a == b) return;
    if (size[a] < size[b]) swap(a, b);
    parent[b] = a;
    size[a] += size[b];
  }
};

struct Prim {
  struct Edge {
    int to, weight;
    Edge(int t, int w) : to(t), weight(w) {}
    bool operator<(const Edge& e) const {
      return weight > e.weight; // min-heap
    }
  };
  int n;
  vector<vector<Edge>> adj;
  vector<bool> visited;
  Prim(int _n) : n(_n) {
    adj.assign(n + 1, vector<Edge>());
    visited.assign(n + 1, false);
  }
  void add_edge(int u, int v, int w, bool directed = false) {
    adj[u].push_back(Edge(v, w));
    if (!directed) adj[v].push_back(Edge(u, w));
  }
  ll mst_cost(int root = 1) {
    ll cost = 0;
    priority_queue<Edge> pq;
    pq.push(Edge(root, 0));
    while (!pq.empty()) {
      Edge e = pq.top();
      pq.pop();
      int u = e.to;
      if (visited[u]) continue;
      visited[u] = true;
      cost += e.weight;
      for (Edge& next : adj[u])
        if (!visited[next.to]) pq.push(next);
    }
    return cost;
  }
};
```

```
struct Node {
  ll sum, pref;
  friend Node operator+(Node l, Node r) {
    return {l.sum + r.sum, max(l.pref, l.sum + r.pref)};
  }
  friend ostream& operator<<(ostream& os, const Node& n) {
    os << "{" << n.sum << ", " << n.pref << "}\n"; return os;
  }
};
struct SGT {
  int n;
  vector<ll> tree;
  SGT(vector<int>& a) {
    n = 1;
    while (n < (int)a.size()) n <<= 1; // hold 2 * next power of 2
    tree.assign(2 * n, 0LL);
    build(a);
    // for (int i = 0; i < (int)a.size(); ++i) change(1, 0, n-1, i, a[i]);
  }
  void build(vector<int>& a) {
    // root node is at 1, children as i*2 and i*2+1
    for (int i = 0; i < (int)a.size(); ++i) {
      tree[n + i] = a[i]; // fill in all the leaf nodes [n, 2*n-1]
    }
    for (int i = n - 1; i >= 1; --i) { // fill in parent nodes
      tree[i] = tree[2 * i] + tree[2 * i + 1];
    }
  }
  // change(1, 0, n-1, idx:0-based, value)
  void change(int node, int node_lo, int node_hi, int idx, ll val) {
    if (node_lo == idx && node_hi == idx) { // leaf node
      tree[node] = val;
      return;
    }
    if (node_lo > idx || node_hi < idx) return;
    int end_low = node_lo + (node_hi - node_lo) / 2;
    change(2 * node, node_lo, end_low, idx, val);
    change(2 * node + 1, end_low + 1, node_hi, idx, val);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
  }
  void change_iter(int idx, ll val) {
    tree[n + idx] = val;
    for (int i = (n + idx) / 2; i >= 1; i /= 2) {
      tree[i] = tree[2 * i] + tree[2 * i + 1];
    }
  }
  ll query_driver(int l, int r) {
    return query(1, 0, n-1, l, r); // make sure l and r are 0-based
  }
  // query(1, 0, n-1, l, r) --> note that lo and hi are all 0-based
  ll query(int node, int node_lo, int node_hi, int query_lo, int query_hi) {
    if (query_lo <= node_lo && node_hi <= query_hi) {
      return tree[node]; // current node's sum
    }
    if (node_lo > query_hi || node_hi < query_lo) {
      return 0; // current node is disjoint in query range
    }
    // new range that the next nodes will encompass
    int end_low = node_lo + (node_hi - node_lo) / 2;
    return query(2 * node, node_lo, end_low, query_lo, query_hi) +
           query(2 * node + 1, end_low + 1, node_hi, query_lo, query_hi);
  }
};
```

Math

```cpp
// classic sieve up to max_n
vector<ll> sieve_up_to(ll max_n) {
  if (max_n < 2) return {};
  vector<bool> is_prime(max_n + 1, true);
  is_prime[0] = is_prime[1] = false;
  for (ll i = 2; i * i <= max_n; ++i) {
    if (is_prime[i]) {
      for (ll j = i * i; j <= max_n; j += i) is_prime[j] = false;
    }
  }
  vector<ll> primes;
  primes.reserve(max_n / log(max_n)); // rough estimate
  for (ll i = 2; i <= max_n; ++i)
    if (is_prime[i]) primes.push_back(i);
  return primes;
}

// segmented sieve for [low, high]
vector<ll> sieve_range(ll low, ll high) {
  if (high < 2 || low > high) return {};
  if (low < 2) low = 2;

  ll limit = sqrt(high) + 1;
  auto base_primes = sieve_up_to(limit);
  vector<bool> is_prime(high - low + 1, true);
  for (ll p : base_primes) {
    ll start = max(p * p, ((low + p - 1) / p) * p);
    for (ll j = start; j <= high; j += p) is_prime[j - low] = false;
  }
  vector<ll> primes;
  for (ll i = 0; i <= high - low; ++i)
    if (is_prime[i]) primes.push_back(low + i);
  return primes;
}

vector<vector<int>> subsets(vector<int>& nums) {
  vector<vector<int>> ans;
  int n = nums.size();
  int len = (1 << n);             // powerset size: 2^n
  for (int i = 0; i < len; ++i) { // every set bits is the index to include
    vector<int> cur;
    for (int j = 0; j < n; ++j) {
      if (i & (1 << j)) cur.push_back(nums[j]);
    }
    ans.push_back(cur);
  }
  return ans;
}

// find the prime factors
vector<int> compute_prime_factors(int n) {
  vector<int> factors;
  for (int i = 2; i * i <= n; ++i) {
    while (n % i == 0) {
      factors.push_back(i);
      n /= i;
    }
  }
  if (n > 1) factors.push_back(n); // n is prime
  return factors;
}
```

```cpp
vector<int> divisors(int n) {
  vector<int> res;
  for (int i = 1; i * i <= n; ++i) {
    if (n % i == 0) {
      res.push_back(i);
      if (i != n / i) res.push_back(n / i);
    }
  }
  sort(res.begin(), res.end());
  return res;
}

ll gcd(ll a, ll b) {
  while (b) {
    a %= b;
    swap(a, b);
  }
  return a;
}
ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }

// Modular Arithmetic
const int MOD = 1e9 + 7;
ll mod_add(ll a, ll b) { return (a + b) % MOD; }
ll mod_sub(ll a, ll b) { return (a - b + MOD) % MOD; }
ll mod_mul(ll a, ll b) { return (a * b) % MOD; }
ll mod_pow(ll a, ll b) {
  ll res = 1;
  while (b) {
    if (b & 1) res = mod_mul(res, a);
    a = mod_mul(a, a);
    b >>= 1;
  }
  return res;
}
// fermats: If p is prime and the gcd(a,p) = 1 then a^p = a mod p
// mod must be prime for fermat's little theorem
ll mod_inv(ll a) { return mod_pow(a, MOD - 2); }
ll mod_div(ll a, ll b) { return mod_mul(a, mod_inv(b)); }

// Combinatorics
const int MAX_N = 1e5 + 10;
vector<ll> fact(MAX_N), inv_fact(MAX_N);
void precompute_factorials(int n) {
  fact[0] = 1;
  for (int i = 1; i <= n; i++) fact[i] = mod_mul(fact[i - 1], i);
  inv_fact[n] = mod_inv(fact[n]);
  for (int i = n - 1; i >= 0; i--) {
    inv_fact[i] = mod_mul(inv_fact[i + 1], i + 1);
  }
}
ll nCr(int n, int r) {
  if (r < 0 || r > n) return 0;
  return mod_mul(fact[n], mod_mul(inv_fact[r], inv_fact[n - r]));
}
ll nPr(int n, int r) {
  if (r < 0 || r > n) return 0;
  return mod_mul(fact[n], inv_fact[n - r]);
}
```

Binary Search

```
// LOWER BOUND
int l = 1, r = n; // both l and r are in the search space
while (l < r) {
  int mid = l + (r - l) / 2;
  // finding the FIRST T in: FFFFFTTTTTTT
  if (cond(mid)) {
    r = mid;
  } else {
    l = mid + 1;
  }
}
return l; // l == r

// UPPER BOUND
int l = 1, r = n; // both l and r are in the search space
while (l < r) {
  int mid = l + (r - l + 1) / 2; // pick right middle element (avoid inf loop)
  // finding the LAST T in: TTTTTTFFFFFF
  if (cond(mid)) {
    l = mid;
  } else {
    r = mid - 1;
  }
}
return l; // l == r
```