Algorithms SWERC 2025

- Structures de données
  - Union Find (Kruskal) ✅
  - Arbre binaire/ segment tree ✅
  - Arbre rouge-noir
- Programmation dynamique (DP)
  - Plus longue sous-séquence croissante ✅
- Graphes
  - Composantes fortement connexes (Kosaraju) ✅
  - Points d'articulation (graphe non orienté) ✅
  - Ponts d'un graphe (arêtes déconnectant le graphe) ✅
  - Bellman-Ford (pour les poids négatifs)
  - Edmonds-Karp (flot maximum) ✅
  - Trouver cycle dans un graphe non-orienté ✅
  - Trouver cycle eulérien ✅
- Géométrie
  - Enveloppe convexe ✅
  - Intersection de deux droites et cross product ✅
  - Deux points les plus proches ✅
- Mathématiques
  - Exponentiation rapide ✅
  - Coefficients de Bézout (ax+by = pgcd(a, b)) ✅
- Chaînes de caractères
  - Rabin-Karp ou KMP ✅
  - Suffix automaton
  - Plus longue sous-séquence commune (DP)
- Autres
  - Tortoise and hare algorithm (cycle dans une liste chaînée)
- Annexe : raccourcis C++ ✅

Structure de données

=> Union Find (complexité des opérations union et find O(1))

```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) { // union avec prise en compte des tailles
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}

int find_set(int v) { // find avec compression de chemin
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

=> Segment tree avec lazy propagation (min segtree + sum range update)

```
struct SegTree {
 ll n;
 vector<ll> data, lazy;

 SegTree(const vector<ll>& v) :
     n(v.size()), data(4 * n), lazy(4 * n, 0) {
   build(1, 0, n - 1, v);
 }

 void build(ll node, ll b, ll e, const vector<ll>& v) {
   if (b == e) {
     data[node] = v[b];
     return;
   }
   ll m = (b + e) / 2;
   build(node * 2, b, m, v);
   build(node * 2 + 1, m + 1, e, v);
   data[node] = min(data[node * 2], data[node * 2 + 1]);
 }
 void add(ll node, ll b, ll e, ll l, ll val) {
   if (e < l) return;
   if (b >= l) {
     lazy[node] += val;
     data[node] += val;
     return;
   }
   ll m = (b + e) / 2;
   add(node * 2, b, m, l, val);
   add(node * 2 + 1, m + 1, e, l, val);
   data[node] = min(data[node * 2], data[node * 2 + 1]) + lazy[node];
 }

 ll pop(ll node, ll b, ll e, ll val) {
   if (b == e) {
     data[node] = 2e18;
     return b;
   }
   ll m = (b + e) / 2;
   val -= lazy[node];

   ll ret = -1;
   if (data[node * 2 + 1] == val)
     ret = pop(node * 2 + 1, m + 1, e, val);
   else ret = pop(node * 2, b, m, val);
   data[node] = min(data[node * 2], data[node * 2 + 1]) + lazy[node];
   return ret;
 }

 void Add(ll l, ll val) { add(1, 0, n - 1, l, val); }
 ll Pop() { return pop(1, 0, n - 1, 0); }
};
```

Programmation dynamique

=> Longest increasing subsequence in O(n log n)

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[l-1] < a[i] && a[i] < d[l])
            d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF)
            ans = l;
    }
    return ans;
}
```

Graphes

=> Kosaraju : composantes fortement connexes en O(n+m)

```
vector<bool> visited; // keeps track of which vertices are already visited

// runs depth first search starting at vertex v.
// each visited vertex is appended to the output vector when dfs leaves it.
void dfs(int v, vector<vector<int>> const& adj, vector<int> &output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

// input: adj -- adjacency list of G
// output: components -- the strongy connected components in G
// output: adj_cond -- adjacency list of G^SCC (by root vertices)
void strongy_connected_components(vector<vector<int>> const& adj,
                                  vector<vector<int>> &components,
                                  vector<vector<int>> &adj_cond) {
    int n = adj.size();
    components.clear(), adj_cond.clear();
    vector<int> order; // will be a sorted list of G's vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end());

    vector<int> roots(n, 0); // gives the root vertex of a vertex's SCC

    // second series of depth first searches
    for (auto v : order)
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            sort(component.begin(), component.end());
            components.push_back(component);
            int root = component.front();
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back(roots[u]);
}
```

=> Articulation points in graph in O(n + m)

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
        visited[v] = true;
        tin[v] = low[v] = timer++;
        int children=0;
        for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
                low[v] = min(low[v], tin[to]);
        } else {
                dfs(to, v);
                low[v] = min(low[v], low[to]);
                if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
                ++children;
        }
        }
        if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
        timer = 0;
        visited.assign(n, false);
        tin.assign(n, -1);
        low.assign(n, -1);
        for (int i = 0; i < n; ++i) {
        if (!visited[i])
                dfs (i);
        }
}
```

=> Bridges in graph in O(n + m)

```
void IS_BRIDGE(int v,int to); // some function to process the found bridge
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
        visited[v] = true;
        tin[v] = low[v] = timer++;
        bool parent_skipped = false;
        for (int to : adj[v]) {
        if (to == p && !parent_skipped) {
                parent_skipped = true;
                continue;
        }
        if (visited[to]) {
                low[v] = min(low[v], tin[to]);
        } else {
                dfs(to, v);
                low[v] = min(low[v], low[to]);
                if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
        }
}

void find_bridges() {
        timer = 0;
        visited.assign(n, false);
        tin.assign(n, -1);
        low.assign(n, -1);
        for (int i = 0; i < n; ++i) {
        if (!visited[i])
                dfs(i);
        }
}
```

=> Finding cycle in directed graph in O(m)

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
        color[v] = 1;
        for (int u : adj[v]) {
        if (color[u] == 0) {
                parent[u] = v;
                if (dfs(u))
                return true;
        } else if (color[u] == 1) {
                cycle_end = v;
                cycle_start = u;
                return true;
        }
        }
        color[v] = 2;
        return false;
}

void find_cycle() {
        color.assign(n, 0);
        parent.assign(n, -1);
        cycle_start = -1;

        for (int v = 0; v < n; v++) {
        if (color[v] == 0 && dfs(v))
                break;
        }

        if (cycle_start == -1) {
        cout << "Acyclic" << endl;
        } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
                cycle.push_back(v);
        cycle.push_back(cycle_start);
        reverse(cycle.begin(), cycle.end());

        cout << "Cycle found: ";
        for (int v : cycle)
                cout << v << " ";
        cout << endl;
        }
```

=> Finding eulerian path in graph in O(m)

```
stack St;
put start vertex in St;
until St is empty
  let V be the value at the top of St;
  if degree(V) = 0, then
       add V to the answer;
       remove V from the top of St;
  otherwise
       find any edge coming out of V;
       remove it from the graph;
       put the second end of this edge in St;
```

=> Maximum flow : Edmond Karp in O(VE^2)

```
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) {
       fill(parent.begin(), parent.end(), -1);
       parent[s] = -2;
       queue<pair<int, int>> q;
       q.push({s, INF});

       while (!q.empty()) {
       int cur = q.front().first;
       int flow = q.front().second;
       q.pop();

       for (int next : adj[cur]) {
              if (parent[next] == -1 && capacity[cur][next]) {
              parent[next] = cur;
              int new_flow = min(flow, capacity[cur][next]);
              if (next == t)
                     return new_flow;
              q.push({next, new_flow});
              }
       }
       }

       return 0;
}

int maxflow(int s, int t) {
       int flow = 0;
       vector<int> parent(n);
       int new_flow;
```

```
    while (new_flow = bfs(s, t, parent)) {
    flow += new_flow;
    int cur = t;
    while (cur != s) {
        int prev = parent[cur];
        capacity[prev][cur] -= new_flow;
        capacity[cur][prev] += new_flow;
        cur = prev;
    }
    }

    return flow;
}
```

Géométrie

=> Enveloppe convexe : Graham scan en O(N log N)

```
struct pt {
        double x, y;
        bool operator == (pt const& t) const {
        return x == t.x && y == t.y;
        }
};

int orientation(pt a, pt b, pt c) {
        double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
        if (v < 0) return -1; // clockwise
        if (v > 0) return +1; // counter-clockwise
        return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
        int o = orientation(a, b, c);
        return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
        pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
        });
        sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
                return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
        });
        if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
        }

        vector<pt> st;
        for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i],
include_collinear))
                st.pop_back();
        st.push_back(a[i]);
        }

        if (include_collinear == false && st.size() == 2 && st[0] == st[1])
        st.pop_back();

        a = st;
}
```

=> Déterminer si deux segment se croisent en O(1)

```
struct pt {
        long long x, y;
        pt() {}
        pt(long long _x, long long _y) : x(_x), y(_y) {}
        pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }
        long long cross(const pt& p) const { return x * p.y - y * p.x; }
        long long cross(const pt& a, const pt& b) const { return (a -
*this).cross(b - *this); }
};

int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }

bool inter1(long long a, long long b, long long c, long long d) {
        if (a > b)
        swap(a, b);
        if (c > d)
        swap(c, d);
        return max(a, c) <= min(b, d);
}

bool check_inter(const pt& a, const pt& b, const pt& c, const pt& d) {
        if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
        return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
                sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}
```

=> Nearest pair of points in O(n log n)

```
double mindist;
pair<int, int> best_pair;

void upd_ans(const pt & a, const pt & b) {
        double dist = sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
        if (dist < mindist) {
        mindist = dist;
        best_pair = {a.id, b.id};
        }
}

vector<pt> t;

void rec(int l, int r) {
        if (r - l <= 3) {
        for (int i = l; i < r; ++i) {
                for (int j = i + 1; j < r; ++j) {
                upd_ans(a[i], a[j]);
                }
        }
        sort(a.begin() + l, a.begin() + r, cmp_y());
        return;
        }

        int m = (l + r) >> 1;
        int midx = a[m].x;
        rec(l, m);
        rec(m, r);

        merge(a.begin() + l, a.begin() + m, a.begin() + m, a.begin() + r,
t.begin(), cmp_y());
        copy(t.begin(), t.begin() + r - l, a.begin() + l);

        int tsz = 0;
        for (int i = l; i < r; ++i) {
        if (abs(a[i].x - midx) < mindist) {
                for (int j = tsz - 1; j >= 0 && a[i].y - t[j].y < mindist; --j)
                upd_ans(a[i], t[j]);
                t[tsz++] = a[i];
        }
        }
}
```

Mathématiques

⇒ PGCD étendu pour trouver (x, y, d) tels que ax+by = d = pgcd(a, b) en O(log a+b)

```
int gcd(int a, int b, int& x, int& y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

Chaînes de caractères

⇒ Algorithme de Rabin Karp (complexité O(|t|) pour trouver un pattern s dans un texte t)

```
vector<int> rabin_karp(string const& s, string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();

    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(T + 1, 0);
    for (int i = 0; i < T; i++)
        h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;
    long long h_s = 0;
    for (int i = 0; i < S; i++)
        h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;

    vector<int> occurrences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i+S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m)
            occurrences.push_back(i);
    }
    return occurrences;
}
```

Annexe : raccourcis C++

```cpp
// Including
#include <cstdio>
#include <iostream>
#include <cmath>
#include <algorithm>
#include <unordered_set>
#include <set>
#include <unordered_map>
#include <map>
#include <vector>
#include <tuple>
#include <stack>
#include <queue>
#include <deque>
#include <bitset>
#include <climits>
#include <complex>
#include <chrono>
#include <random>

using namespace std;

// STL functions
#define pb push_back
#define mt make_tuple
#define mp make_pair
#define fi first
#define se second

// Iteration
#define all(c) (c).begin(), (c).end()
#define sz(x) (int)(x).size()
#define fo(i,n) for(int i=0; i<n; i++)

// Input and output
#define si(x) scanf("%d", &x)
#define sl(x) scanf("%lld", &x)
#define ss(x) getline(cin, x)
#define pi(x) printf("%d\n", x)
#define pl(x) printf("%lld\n", x)
#define ps(x) cout << x << "\n"

// Types
using ll = long long;
using ld = long double;
using uint = unsigned int;
using ull = unsigned long long;
using pii = pair<int, int>;
using pll = pair<ll, ll>;
using vi = vector<int>;
using vvi = vector<vector<int>>;
using vll = vector<ll>;

// Debugging
#define isDebug true
#ifdef isDebug
#define debug(x) cout << #x << "=" << x << "\n"
#else
#define debug(x)
#endif

int main() {
    ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    return 0;
}
```