Nombre del equipo

CUNEF Universidad

Integrante 1 Integrante 2 Integrante 3

Chuletario ICPC CUNEF

Binary search

time: O(log n), space: O(1) | Encontrar un valor en un array ordenado.

```cpp
#include <vector>
using namespace std;
int binary_search_idx(const vector<int>& a, int target) {
    int lo = 0, hi = (int)a.size() - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == target) return mid;
        if (a[mid] < target) lo = mid + 1;
        else                 hi = mid - 1;
    }
    return -1;
}
```

Fast I/O (C++)

```cpp
// C++ fast I/O -- put these two lines first, inside main():
ios::sync_with_stdio(false);
cin.tie(nullptr);
// Then read/print as usual:  cin >> x;   cout << x << '\n';
// (use '\n', never endl inside loops -- endl flushes the buffer every time).
```

Fast I/O (Python)

```python
# Python fast I/O -- swap input() for the buffered reader (drop-in, at the top):
import sys
input = sys.stdin.readline
# input().split() reads one line; for many tokens at once: sys.stdin.read().split().
# Print a lot in one go:  print("\n".join(map(str, answers)))
```

Dijkstra

time: O((V+E) log V), space: O(V+E) | Camino más corto desde un origen con pesos no negativos.

```cpp
#include <vector>
#include <queue>
#include <functional>
using namespace std;
const long long INF = 1e18;
vector<long long> dijkstra(int n, const vector<vector<pair<int, int>>>& adj, int src) {
    vector<long long> dist(n, INF);
    priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<>> pq;
    dist[src] = 0;
    pq.push({0, src});
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (d > dist[u]) continue;
        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

MiniMax

time: O(n), space: O(n) | Encontrar el mejor movimiento en un juego de dos jugadores.

```cpp
#include <vector>
#include <functional>
#include <limits>
#include <utility>
using namespace std;
template <typename State>
pair<double, int> minimax(
    const State& current_state,
    int maximizing_player,
    const function<vector<State>(const State&)>& possible_moves_funct,
    const function<double(const State&)>& score_funct) {
    vector<State> next_states = possible_moves_funct(current_state);
    if (next_states.empty())
        return {score_funct(current_state), -1};
    double best_value = (maximizing_player == 1)
        ? -numeric_limits<double>::infinity()
        :  numeric_limits<double>::infinity();
    int best_move = -1;
    for (int index = 0; index < (int) next_states.size(); ++index) {
        double value = minimax(next_states[index], -maximizing_player,
                               possible_moves_funct, score_funct).first;
        if (maximizing_player == 1) {
            if (value > best_value) { best_value = value; best_move = index; }
        } else {
            if (value < best_value) { best_value = value; best_move = index; }
        }
    }
    return {best_value, best_move};
}
```

Fenwick Tree (BIT)

build: O(n), update: O(log n), query: O(log n), space: O(n) | Sumas de prefijos con actualizaciones puntuales.

```cpp
#include <vector>
using namespace std;
struct FenwickTree {
    int n;
    vector<int> tree;
    FenwickTree(int n) : n(n), tree(n) {}
    void update(int index, int delta) {
        for (index++; index <= n; index += index & -index)
            tree[index - 1] += delta;
    }
    int query(int index) {
        int acc = 0;
        for (; index > 0; index -= index & -index)
            acc += tree[index - 1];
        return acc;
    }
    int query_range(int l, int r) { return query(r) - query(l); }
};
```

Convex hull

time: O(n log n), space: O(n) | El menor polígono convexo que contiene un conjunto de puntos.

```cpp
#include <vector>
#include <algorithm>
using namespace std;
struct P {
    long long x, y;
    bool operator<(const P& o) const {
        return x < o.x || (x == o.x && y < o.y);
    }
};
long long cross(const P& O, const P& A, const P& B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}
vector<P> convex_hull(vector<P> pts) {
    int n = pts.size();
    if (n < 3) return pts;
    sort(pts.begin(), pts.end());
    vector<P> hull;
    for (int i = 0; i < n; i++) {
        while (hull.size() >= 2 &&
               cross(hull[hull.size() - 2], hull.back(), pts[i]) <= 0)
            hull.pop_back();
        hull.push_back(pts[i]);
    }
    size_t base = hull.size() + 1;
    for (int i = n - 2; i >= 0; i--) {
        while (hull.size() >= base &&
               cross(hull[hull.size() - 2], hull.back(), pts[i]) <= 0)
            hull.pop_back();
        hull.push_back(pts[i]);
    }
    hull.pop_back();
    return hull;
}
```