Soluciones OIFem II Nivel 1

Concurso de práctica 2

Majority Element

Teoría

- Mapas

Solución

Creamos un mapa con las frecuencias de cada elemento y comprobamos cuál es el mayoritario.

Código

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        map<int, int> occ;
        for (int x: nums)
            occ[x]++;
        int n = (int) nums.size();
        for (map<int, int>::iterator it = occ.begin(); it != occ.end(); it++)
            if (n/2 < it->second)
                return it->first;
        return 0;
    }
};
```

K Closest Points to Origin

Teoría

- Priority queues

Solución

Creamos una PQ donde guardamos los puntos, ordenados por distancia al punto $(0, 0)$ y retornamos los k primeros.

Código

```cpp
class Solution {
public:
    vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {
        priority_queue<pair<int, size_t>> PQ;
        for (size_t i = 0; i < points.size(); i++) {
            PQ.push(make_pair(-points[i][0]*points[i][0] -
            ↪  points[i][1]*points[i][1], i));
        }
        vector<vector<int>> ans(k);
        for (int i = 0; i < k; i++) {
            size_t ind = PQ.top().second;
            PQ.pop();
            ans[i] = points[ind];
        }
        return ans;
    }
};
```

Roman to Integer

Teoría

- Implementación

Solución

Para resolver este problema, bastaba con seguir cuidadosamente las instrucciones del enunciado.

Código

```cpp
class Solution {
public:
    int romanToInt(string s) {
        int n = (int) s.length();
        if (n == 0) return 0;
```

```
        int i = 0;
        int ans = 0;
        while(i < n) {
            if (s[i] == 'M') {
                ans += 1000;
                i++;
            } else if (s[i] == 'D') {
                ans += 500;
                i++;
            } else if (s[i] == 'C') {
                if (i < n-1 && s[i+1] == 'D') {
                    ans += 400;
                    i += 2;
                } else if (i < n-1 && s[i+1] == 'M') {
                    ans += 900;
                    i += 2;
                } else {
                    ans += 100;
                    i++;
                }
            } else if (s[i] == 'L') {
                ans += 50;
                i++;
            } else if (s[i] == 'X') {
                if (i < n-1 && s[i+1] == 'L') {
                    ans += 40;
                    i += 2;
                } else if (i < n-1 && s[i+1] == 'C') {
                    ans += 90;
                    i += 2;
                } else {
                    ans += 10;
                    i++;
                }
            } else if (s[i] == 'V') {
                ans += 5;
                i++;
            } else if (i < n-1 && s[i+1] == 'V') {
                ans += 4;
                i += 2;
            } else if (i < n-1 && s[i+1] == 'X') {
                ans += 9;
                i += 2;
            } else {
                ans++;
                i++;
            }
        }
        return ans;
    }
};
```

Climbing Stairs

Teoría

- Programación dinámica

Solución

Si nos fijamos, la solución son los números Fibonacci. Por lo tanto, podemos usar la DP que aprendimos para encontrarlos.

Código

```cpp
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 1) {
            if (n == 0)
                return 0;
            return 1;
        }
        int prevprev = 1, prev = 2;
        for (int i = 0; i < n-2; i++) {
            int cur = prev + prevprev;
            prevprev = prev;
            prev = cur;
        }
        return prev;
    }
};
```

Word Search

Teoría

- Backtracking
- DFS

Solución

Hacemos una DFS con backtracking para tratar de encontrar la palabra y evitar así ciclos.

Código

```cpp
class Solution {
public:
    vector<vector<bool>> vis;
    bool DFS(int ind, int i, int j, string &word, vector<vector<char>>& board) {
```

```
        if (ind == (int) word.size()) return true;
        if (i < 0 || j < 0 || i == (int) board.size() || j == (int)
        ↪  board[0].size()) return false;
        if (word[ind] != board[i][j] || vis[i][j]) return false;
        vis[i][j] = true;
        if (DFS(ind+1, i-1, j, word, board)) {
            vis[i][j] = false;
            return true;
        }
        if (DFS(ind+1, i+1, j, word, board)) {
            vis[i][j] = false;
            return true;
        }
        if (DFS(ind+1, i, j+1, word, board)) {
            vis[i][j] = false;
            return true;
        }
        if (DFS(ind+1, i, j-1, word, board)) {
            vis[i][j] = false;
            return true;
        }
        vis[i][j] = false;
        return false;
    }
    bool correct(int startR, int startC, string &word, vector<vector<char>>&
    ↪  board) {
        int R = (int) board.size(), C = (int) board[0].size();
        vis.assign(R, vector<bool>(C, false));
        return DFS(0, startR, startC, word, board);
    }
    bool exist(vector<vector<char>>& board, string word) {
        int R = (int) board.size(), C = (int) board[0].size();
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                if (board[i][j] == word[0] && correct(i, j, word, board)) {
                    return true;
                }
            }
        }
        return false;
    }
};
```

All Paths from Source to Target

Teoría

- DFS
- Backtracking

Solución

Hacemos DFS con backtracking para encontrar todos los caminos, guardándolos a medida que los vamos encontrando.

Código

```
class Solution {
public:
    vector<vector<vector<int>>> pathsToNode;
    vector<int> curPath;
    void dfs(int node, vector<vector<int>>& graph) {
        curPath.push_back(node);
        pathsToNode[node].push_back(curPath);
        for (int neigh: graph[node])
            dfs(neigh, graph);
        curPath.pop_back();
    }
    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        pathsToNode = vector<vector<vector<int>>>(graph.size());
        dfs(0, graph);
        return pathsToNode[(int)graph.size()-1];
    }
};
```