Soluciones OIFem II Nivel 2

Entrenamiento 5

Maximum Product of Two Elements in an Array

Teoría

- Vectores

Solución

El mayor producto será el producto de los dos mayores índices positivos menos uno o de los dos negativos con mayor valor absoluto menos uno. Iteraremos sobre el vector, llevando la cuenta en todo momento de los dos positivos y dos negativos con mayor absoluto que hayamos visto. Finalmente, retornaremos el mayor de estos dos productos.

Código

```cpp
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int menor, segundoMenor, mayor, segundoMayor;
        if (nums[0] < nums[1]) {
            segundoMayor = menor = nums[0];
            mayor = segundoMenor = nums[1];
        } else {
            segundoMayor = menor = nums[1];
            mayor = segundoMenor = nums[0];
        }
        for (size_t i = 2; i < nums.size(); i++) {
            if (nums[i] < menor) {
                segundoMenor = menor;
                menor = nums[i];
            } else if (nums[i] < segundoMenor)
                segundoMenor = nums[i];
            if (nums[i] > mayor) {
                segundoMayor = mayor;
                mayor = nums[i];
            } else if (nums[i] > segundoMayor)
                segundoMayor = nums[i];
        }
```

```
24            return max((menor-1)*(segundoMenor-1), (mayor-1)*(segundoMayor-1));
25        }
26    };
```

Next Greater Element I

Teoría

- Stacks
- Unordered maps

Solución

Podemos usar un stack para guardar, para una posición i, `nums2[i]`, seguido de `nums2[j]`, donde j es el menor índice mayor que i tal que `nums2[i] < nums2[j]`. Después hacemos lo mismo para j y así hasta llegar al final de `nums2`. Iteramos entonces sobre `nums2` en orden reverso y vamos actualizando el stack cada vez que llegamos a un nuevo índice. Podemos usarlo para calcular para cada índice de `nums2` el siguiente índice que tenga un número mayor (o -1 si este no existe). Si además creamos un mapa desordenado donde guardemos, para cada número de `nums2`, el índice donde ocurre, podemos encontrar la solución con un bucle sencillo.

Código

```cpp
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        int n = (int) nums1.size(), m = (int) nums2.size();
        // Creamos un stack como el de la descripción
        stack<int> S;
        S.push(nums2[m-1]); // inicialmente solo meteremos el último número de la lista
        vector<int> nextGreater(m);
        nextGreater[m-1] = -1; // la última posición no tiene números después y por tanto la respuesta será -1 en ese caso
        unordered_map<int, int> position;
        position[nums2[m-1]] = m-1;

        for (int i = m-2; i >= 0; i--) {
            position[nums2[i]] = i;
            while(S.size() && S.top() <= nums2[i])
                S.pop();
            if (S.empty())
                nextGreater[i] = -1;
            else
                nextGreater[i] = S.top();
            S.push(nums2[i]);
        }

        vector<int> ans(n);
        for (size_t i = 0; i < ans.size(); i++)
            ans[i] = nextGreater[position[nums1[i]]];
        return ans;
    }
};
```

Check Whether Two Strings are Almost Equivalent

Teoría

- Unordered maps

Solución

Creamos dos mapas desordenados, `freq1` y `freq2`, donde guardamos cuántas ocurrencias hay de cada carácter en las dos palabras. Una vez hecho esto, iteramos sobre todas las letras minúsculas y vemos si hay alguna que incumple la regla impuesta, retornando `false` en ese caso. Si no, retornamos `true`.

Código

```cpp
class Solution {
public:
    bool checkAlmostEquivalent(string word1, string word2) {
        unordered_map<char, int> freq1, freq2;
        for (char c = 'a'; c <= 'z'; c++)
            freq1[c] = freq2[c] = 0;
        for (char c: word1)
            freq1[c]++;
        for (char c: word2)
            freq2[c]++;
        for (char c = 'a'; c <= 'z'; c++)
            if (abs(freq1[c] - freq2[c]) > 3)
                return false;
        return true;
    }
};
```

Valid Anagram

Teoría

- Unordered maps

Solución

Cambiamos la solución al problema anterior para comprobar que la frecuencia de un carácter en s es la misma que la del carácter en t.

Código

```cpp
class Solution {
public:
    bool checkAlmostEquivalent(string word1, string word2) {
        unordered_map<char, int> freq1, freq2;
        for (char c = 'a'; c <= 'z'; c++)
            freq1[c] = freq2[c] = 0;
        for (char c: word1)
            freq1[c]++;
        for (char c: word2)
            freq2[c]++;
        for (char c = 'a'; c <= 'z'; c++)
            if (abs(freq1[c] - freq2[c]) > 3)
                return false;
        return true;
    }
};
```

First Unique Character in a String

Teoría

- Unordered maps

Solución

Creamos un mapa donde, cada vez que descubramos un carácter nuevo en s, guardaremos la posición donde esto ocurre. Sin embargo, si vemos otra ocurrencia de ese carácter, guardaremos -1 en su lugar en el mapa. Finalmente, iteraremos sobre los carácteres de s empezando por el principio y retornaremos el primero cuyo índice en el mapa no sea -1.

Código

```
class Solution {
public:
    int firstUniqChar(string s) {
        unordered_map <char, int> M;

        for (int i = 0; i < s.size(); ++i){
            char letra = s[i];

            if (M.find(letra) == M.end()) M[letra] = i;
            else M[letra] = -1;
        }

        for (char c: s){
           if (M[c] != -1) return M[c];
        }

        return -1;
    }
};
```

Find the Difference

Teoría

- Unordered maps

Solución

La solución a este problema es casi idéntica a la del de los anagramas. Excepto que en este caso, en cuanto vemos una diferencia en frecuencias, sabemos que esa es la letra añadida y la retornamos.

Código

```cpp
class Solution {
public:
    char findTheDifference(string s, string t) {
        unordered_map<char, int> freq1, freq2;
        for (char c = 'a'; c <= 'z'; c++)
            freq1[c] = freq2[c] = 0;
        for (char c: s)
            freq1[c]++;
        for (char c: t)
            freq2[c]++;
        for (char c = 'a'; c < 'z'; c++)
            if (freq1[c] != freq2[c])
                return c;
        return 'z';
    }
};
```

Kth Largest Element in a Stream

Teoría

- Multisets

Solución

Resolveremos este problema con un multiset donde en todo momento estarán guardados los K mayores números de los que hayamos leído. Cada vez que leemos un nuevo número, lo añadiremos al set, eliminando el menor de los $K+1$ en él. Siempre el K-ésimo más grande será el menor del set y será el que retornaremos desde `add()`.

Código

```
class KthLargest {
public:
    multiset<int> S;
    int K;
    KthLargest(int k, vector<int>& nums) {
        K = k;
        for (int n: nums) {
            S.insert(n);
            if (S.size() > k)
                S.erase(S.begin());
        }
    }

    int add(int val) {
        S.insert(val);
        if (S.size() > K)
            S.erase(S.begin());
        return *S.begin();
    }
};
```

Remove Outermost Parentheses

Teoría

- Stacks

Solución

La solución consiste en descomponer s de forma primitiva y quitar el primer y último carácter de cada una de sus partes. Luego, retornaremos la concatenación de esas partes.

Para encontrar la descomposición prima, usaremos un stack donde guardaremos los paréntesis que hemos visto abiertos sin cerrar aún. Cada vez que veamos un paréntesis de cierre, quitaremos el que esté abierto arriba del todo del stack. Si en algún momento el stack queda vacío, sabremos que hemos llegado al final de una parte y la guardaremos. Como en este caso el stack solo guarda paréntesis abiertos, podemos prescindir de usarlo y sustituirlo por una variable `opening`.

Código

```cpp
class Solution {
public:
    vector<string> primitiveDecomposition(string & s) {
        size_t opening = 0, curLen = 0;
        vector<string> ret;
        for (size_t i = 0; i < s.size(); i++) {
            curLen++;
            if (s[i] == ')') {
                opening--;
                if (opening == 0) {
                    ret.push_back(s.substr(i-curLen+1, curLen));
                    curLen = 0;
                }
            } else opening++;
        }
        return ret;
    }

    string removeOuterParentheses(string s) {
        vector<string> P = primitiveDecomposition(s);
        string ret = "";
        for (string par: P)
            ret += par.substr(1, par.length()-2);
        return ret;
    }
};
```

The K Weakest Rows in a Matrix

Teoría

- Sets

Solución

Crearemos un conjunto ordenado de parejas. Iteraremos sobre las filas de la matriz, para cada fila guardando una pareja con formato (número de soldados, índice de la fila). La solución será el índice de las primeras k parejas del conjunto. Por lo tanto, podremos ir quitando parejas del conjunto cuando se pase de tamaño.

Código

```cpp
class Solution {
public:
    vector<int> kWeakestRows(vector<vector<int>>& mat, int k) {
        set<pair<int, int>> S;
        int m = (int) mat.size();
        for (int i = 0; i < m; i++) {
            int numSoldiers = count(mat[i].begin(), mat[i].end(), 1);
            S.insert(make_pair(numSoldiers, i));
            if (S.size() > k) {
                set<pair<int, int>>::iterator it = S.end();
                it--;
                S.erase(it);
            }
        }
        vector<int> ans(k);
        while(S.size()) {
            ans[k - S.size()] = (*S.begin()).second;
            S.erase(S.begin());
        }
        return ans;
    }
};
```

Find the Winner of the Circular Game

Teoría

- Queues

Solución

Para resolver este problema, basta con simular el proceso descrito con un `queue`.

Código

```cpp
class Solution {
public:
    int findTheWinner(int n, int k) {
        queue<int> Q;
        for (int i = 1; i <= n; i++)
            Q.push(i);
        while(Q.size() > 1) {
            for (int i = 0; i < k-1; i++) {
                Q.push(Q.front());
                Q.pop();
            }
            Q.pop();
        }
        return Q.front();
    }
};
```

Reduce Array Size to The Half

Teoría

- Unordered maps
- Vectores

Solución

Crearemos inicialmente un vector donde guardemos la frecuencia en el array de cada elemento distinto en él. Luego, guardaremos estas parejas de (elemento, frecuencia) en un vector y lo ordenaremos por la frecuencia. Iteraremos de más frecuente a menos frecuente, guardando el total de las frecuencias hasta que estas lleguen a la mitad de la longitud del array.

Código

```
class Solution {
public:
    int minSetSize(vector<int>& arr) {
        unordered_map<int, int> occ;
        for (int n: arr)
            occ[n]++;

        vector<pair<int, int>> parejas;
        for (auto it: occ)
            parejas.push_back(make_pair(it.second, it.first));
        sort(parejas.begin(), parejas.end());

        int total = 0;
        for (int i = (int) parejas.size()-1; i > 0; i--) {
            total += parejas[i].first;
            if (total *2 >= arr.size())
                return parejas.size()-i;
        }
        return parejas.size();
    }
};
```

132 Pattern

Teoría

- Sets

Solución

Crearemos inicialmente un vector donde guardemos, en la i-ésima posición el índice j que guarde el menor número de `nums[0..i]`. En caso de empate, se guardará el menor índice que contenga ese valor, por lo que `prefMin[i] = i` únicamente en el caso de que el menor valor de `nums[0..i-1]` sea estrictamente mayor que `nums[i]`.

Tras calcular este vector, crearemos un conjunto `vistos`, con los números vistos hasta el momento, e iteraremos sobre `nums` en orden reverso. Para cada índice i que visitemos, cogeremos el menor número del prefijo de i y comprobaremos si existe algún número entre el menor del prefijo y `nums[i]` en `vistos`. Si se da este caso y `nums[i]` no es el menor de su prefijo, retornaremos `true`.

Código

```
class Solution {
public:
    bool find132pattern(vector<int> & nums) {
        int n = (int) nums.size();
        if (n < 3)
            return false;

        vector<int> prefMin(n); // prefMin[i] = índice del menor elemento en
        ↪  nums[0..i]
        // En caso de empate será el índice menor, lo que significa que si
        ↪  prefMin[i] = i, nums[i] es estrictamente menor que cualquier número en
        ↪  nums[0..i-1]
        prefMin[0] = 0;
        for (int i = 1; i < n; i++) {
            if (nums[i] < nums[prefMin[i-1]]) prefMin[i] = i;
            else prefMin[i] = prefMin[i-1];
        }

        set<int> vistos;
        vistos.insert(nums[n-1]);
        for (int i = n-2; i > 0; i--) {
            // descartaremos encontrar un 132 con nums[i] si nums[i] es el menor
            ↪  de su prefijo o si el mayor visto hasta ahora no es mayor que el
            ↪  menor del prefijo de nums[i]
            if (prefMin[i] == i || *vistos.rbegin() <= nums[prefMin[i]]) {
                vistos.insert(nums[i]);
                continue;
            }

            // comprobaremos si hay algún número en vistos que esté entre
            ↪  prefMin[i] y nums[i]
            int cur = *vistos.lower_bound(nums[prefMin[i]]+1);
            if (cur < nums[i])
```

```
                return true;
            vistos.insert(nums[i]);
        }
        return false;
    }
};
```

Minimum Remove to Make Valid Parentheses

Teoría

- Sets
- Stacks

Solución

Iteraremos sobre los carácteres de s en orden, guardando en un stack los índices de los paréntesis de apertura en todo momento. Cada vez que veamos uno de cierre, hay dos opciones: hay uno ahora mismo de apertura que este puede cerrar: en tal caso consideraremos que este cierra el paréntesis más cercano y lo quitaremos del stack; no hay ninguno de apertura: guardaremos el índice actual en un set con los paréntesis de cierra sobrantes.

Tras esto, si quedan paréntesis abiertos sin cerrar, los añadiremos al conjunto de los paréntesis sobrantes. Finalmente, la solución será s sin los paréntesis sobrantes.

Código

```
class Solution {
public:
    string minRemoveToMakeValid(string s) {
        stack<int> opening_indices;
        set<int> unmatched;
        for (int i = 0; i < (int) s.length(); i++) {
            if (s[i] == '(')
                opening_indices.push(i);
            else if (s[i] == ')') {
                if (opening_indices.size())
                    opening_indices.pop();
                else
                    unmatched.insert(i);
            }
        }

        while(opening_indices.size()) {
            unmatched.insert(opening_indices.top());
            opening_indices.pop();
        }

        string ret = "";

        for (int i = 0; i < (int) s.length(); i++) {
            if (unmatched.find(i) == unmatched.end())
                ret.push_back(s[i]);
        }
        return ret;
    }
};
```

Trapping Rain Water

Teoría

- Sets
- Stacks

Solución

Para todo i, la altura máxima que podrá tener de lluvia la i-ésima posición será igual al menor de los siguientes dos valores: la elevación máxima de un índice menor que i y la elevación máxima de un índice mayor que i. De una forma similar a lo que hemos hecho en problemas anteriores, podremos encontrar estos dos valores. Sumaremos al total de lluvia la diferencia entre este valor y la elevación actual, en caso de que esta sea menor y por tanto pueda albergar lluvia.

Código

```
class Solution {
public:
    int trap(vector<int> & height) {
        int n = (int) height.size();
        if (n == 0) return 0;

        vector<int> right, left;
        right = left = vector<int>(n);
        right[n-1] = -1;
        for (int i = n-2; i >= 0; i--)
            right[i] = max(right[i+1], height[i+1]);
        left[0] = -1;
        for (int i = 1; i < n; i++)
            left[i] = max(left[i-1], height[i-1]);

        int ret = 0;
        for (int i = 0; i < n; i++) {
            if (left[i] == -1 || right[i] == -1) continue;
            ret += max(0, min(left[i], right[i]) - height[i]);
        }
        return ret;
    }
};
```

The Skyline Problem

Teoría

- Merge sort
- Vectores

Solución

Haremos un algoritmo de tipo *divide y vencerás*, modificando *merge sort*. Recursivamente, partiremos la lista que se nos da de edificios en dos mitades, haremos el skyline de las dos mitades y juntaremos ambas con una función `merge()` donde hay que fijarse bien en todos los casos que pueden darse. En este problema es muy importante la implementación ordenada y bien estructurada, ya que es fácil que haya bugs de lógica por casos que una no contempla.

Código

```
class Solution {
public:
    void addBuilding(vector<vector<int>> & answer, vector<int> & building) {
        int n = (int) answer.size();
        if (building[0] >= answer[n-1][1]) {
            // no hay overlap
            answer.push_back(building);
        } else if (answer[n-1][2] == building[2]) {
            // son de la misma altura
            answer[n-1][1] = max(building[1], answer[n-1][1]);
        } else if (answer[n-1][0] == building[0]) {
            // empiezan a la vez
            if (building[2] == answer[n-1][2])
                answer[n-1][1] = max(answer[n-1][1], building[1]);
            else if (building[2] > answer[n-1][2]) {
                if (answer[n-1][1] > building[1])
                    answer.push_back({building[1], answer[n-1][1],
                    ↪  answer[n-1][2]});
                answer[n-1][1] = building[1];
                answer[n-1][2] = building[2];
            } else {
                if (building[1] > answer[n-1][1])
                    answer.push_back({answer[n-1][1], building[1], building[2]});
            }
        } else if (answer[n-1][2] > building[2]) {
            // el último es más alto que el actual
            if (building[1] > answer[n-1][1]) {
                answer.push_back({answer[n-1][1], building[1], building[2]});
            }
        } else {
            // el último es más bajo que el actual
            answer.push_back(building);
            if (answer[n-1][1] > building[1])
                answer.push_back({building[1], answer[n-1][1], answer[n-1][2]});
            answer[n-1][1] = building[0];
```

```
        }
    }

    vector<vector<int>> merge(vector<vector<int>> & left, vector<vector<int>> &
    ↪  right) {
        vector<vector<int>> answer;
        size_t i = 0, j = 0;
        if (left[0][0] != right[0][0]) {
            if (left[0][0] < right[0][0]) {
                answer.push_back(left[0]);
                i++;
            } else {
                answer.push_back(right[0]);
                j++;
            }
        } else if (left[0][2] > right[0][2]) {
            answer.push_back(left[0]);
            i++;
        } else {
            answer.push_back(right[0]);
            j++;
        }

        while(i < left.size() && j < right.size()) {
            if (left[i][0] != right[j][0]) {
                if (left[i][0] < right[j][0]) {
                    addBuilding(answer, left[i]);
                    i++;
                } else {
                    addBuilding(answer, right[j]);
                    j++;
                }
            } else {
                addBuilding(answer, left[i]);
                i++;
            }
        }
        while(i < left.size()) {
            addBuilding(answer, left[i]);
            i++;
        }
        while(j < right.size()) {
            addBuilding(answer, right[j]);
            j++;
        }
        return answer;
    }

    vector<vector<int>> construct(vector<vector<int>>& buildings, int st, int end)
    ↪  {
        if (st == end)
            return {buildings[st]};
        int mid = st + (end-st)/2;
        vector<vector<int>> left = construct(buildings, st, mid);
        vector<vector<int>> right = construct(buildings, mid+1, end);
```

```
88          return merge(left, right);
89      }
90
91      vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
92          vector<vector<int>> full = construct(buildings, 0, (int)
            ↪  buildings.size()-1);
93          vector<vector<int>> answer;
94          for (size_t i = 0; i < full.size(); i++) {
95              if (answer.empty() || answer[answer.size()-1][1] != full[i][2])
96                  answer.push_back({full[i][0], full[i][2]});
97              if (i+1 == full.size() || full[i+1][0] > full[i][1])
98                  answer.push_back({full[i][1], 0});
99          }
100         return answer;
101     }
102 };
```