# Algoritmo Union-Find en C++

El algoritmo Union-Find es una estructura de datos que proporciona métodos eficientes para gestionar un conjunto de elementos divididos en una colección de subconjuntos disjuntos. Es especialmente útil para problemas de conectividad, donde necesitamos saber si dos elementos pertenecen al mismo conjunto.

## Conceptos Clave

- **Find**: Determina a qué subconjunto pertenece un elemento particular. Esto se puede hacer utilizando un identificador de conjunto, como el representante de cada subconjunto.
- **Union**: Une dos subconjuntos en un único subconjunto.

### Implementación Básica

La estructura de datos Union-Find se implementa utilizando dos estructuras principales:

- Un array `parent` donde `parent[i]` representa el padre del nodo `i`. Si `parent[i] = i`, entonces `i` es el representante (o raíz) de su conjunto.
- Un array `rank` que se utiliza para optimizar la operación de unión, evitando aumentar la altura del árbol de subconjuntos.

### Código de la Implementación Básica

```
#include <iostream>
#include <vector>

class UnionFind {
private:
    std::vector<int> parent, rank;

public:
    UnionFind(int n) : parent(n), rank(n, 0) {
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }

    int find(int p) {
        if (parent[p] != p) {
            parent[p] = find(parent[p]);
        }
        return parent[p];
    }

    void unite(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);

        if (rootP != rootQ) {
            if (rank[rootP] > rank[rootQ]) {
```

```
                parent[rootQ] = rootP;
            } else if (rank[rootP] < rank[rootQ]) {
                parent[rootP] = rootQ;
            } else {
                parent[rootQ] = rootP;
                rank[rootP]++;
            }
        }
    }
};
```

### Ejemplo de Uso Básico

```
int main() {
    UnionFind uf(10);

    uf.unite(1, 2);
    uf.unite(3, 4);
    uf.unite(1, 4);

    std::cout << "1 y 4 están conectados: " << (uf.find(1) == uf.find(4)) <<
std::endl;
    std::cout << "1 y 5 están conectados: " << (uf.find(1) == uf.find(5)) <<
std::endl;

    return 0;
}
```

**Salida esperada:**

```
1 y 4 están conectados: 1
1 y 5 están conectados: 0
```

## Aplicaciones del Algoritmo Union-Find

### 1. Detección de Ciclos en un Grafo No Dirigido

El algoritmo Union-Find se puede usar para detectar ciclos en un grafo no dirigido. Si durante el proceso de unión de dos vértices encontramos que ambos ya pertenecen al mismo conjunto, entonces el grafo contiene un ciclo.

```
#include <iostream>
#include <vector>
#include <tuple>

bool hasCycle(int n, const std::vector<std::tuple<int, int>>& edges) {
    UnionFind uf(n);
    for (const auto& edge : edges) {
        int u, v;
        std::tie(u, v) = edge;
        if (uf.find(u) == uf.find(v)) {
            return true;
```

```
        }
        uf.unite(u, v);
    }
    return false;
}

int main() {
    std::vector<std::tuple<int, int>> edges = {{0, 1}, {1, 2}, {2, 0}, {3, 4}};
    if (hasCycle(5, edges)) {
        std::cout << "El grafo tiene un ciclo." << std::endl;
    } else {
        std::cout << "El grafo no tiene un ciclo." << std::endl;
    }
    return 0;
}
```

**Salida esperada:** `El grafo tiene un ciclo.`

### 2. Agrupamiento Jerárquico (Clustering)

El algoritmo Union-Find se puede usar en técnicas de agrupamiento jerárquico para unir clústeres cercanos hasta formar un número específico de clústeres.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int n = 5;
    UnionFind uf(n);
    std::vector<std::tuple<int, int, int>> edges = {{0, 1, 1}, {1, 2, 2}, {2, 3, 3},
{3, 4, 4}, {0, 4, 5}};

    std::sort(edges.begin(), edges.end(), [](auto& a, auto& b) {
        return std::get<2>(a) < std::get<2>(b);
    });

    int clusters = n;
    for (const auto& [u, v, w] : edges) {
        if (uf.find(u) != uf.find(v)) {
            uf.unite(u, v);
            clusters--;
        }
        if (clusters == 2) break;
    }

    std::cout << "Número de clústeres: " << clusters << std::endl;
    return 0;
}
```

**Salida esperada:** `Número de clústeres: 2`

### 3. Componentes Conectados en un Grafo

Este algoritmo es útil para encontrar todos los componentes conectados en un grafo no dirigido, agrupando vértices que están directamente o indirectamente conectados entre sí.

```
#include <iostream>
#include <vector>
#include <unordered_map>

int main() {
    UnionFind uf(7);
    std::vector<std::pair<int, int>> edges = {{0, 1}, {1, 2}, {3, 4}, {5, 6}};

    for (const auto& edge : edges) {
        uf.unite(edge.first, edge.second);
    }

    std::unordered_map<int, std::vector<int>> components;
    for (int i = 0; i < 7; ++i) {
        int root = uf.find(i);
        components[root].push_back(i);
    }

    for (const auto& [root, component] : components) {
        std::cout << "Componente con raíz " << root << ": ";
        for (int node : component) {
            std::cout << node << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

**Salida esperada:**

```
Componente con raíz 0: 0 1 2
Componente con raíz 3: 3 4
Componente con raíz 5: 5 6
```