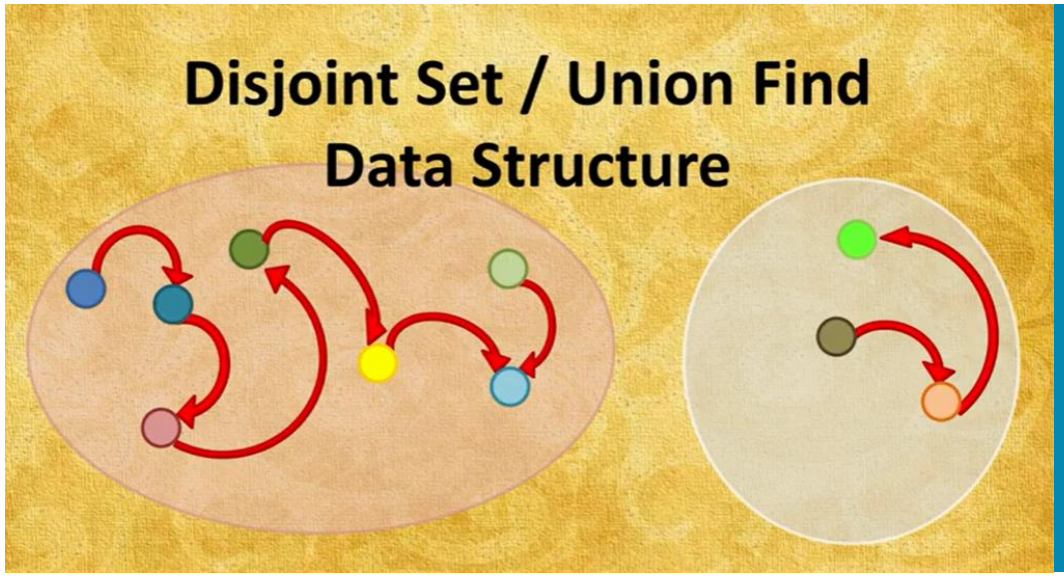



# Complejidad Algorítmica

**Unidad 2: Algoritmos voraces, programación dinámica y problemas P-NP**

**Módulo 9: Estructura de datos para conjuntos disjuntos (UFDS)**

# Complejidad Algorítmica

Semana 9 / Sesión 1

**MÓDULO 9: Estructura de datos para conjuntos disjuntos (UFDS)**

## Contenido

1. Definición de Union-Find Disjoint Sets (UFDS)
2. Algoritmos o Estrategias Union-Find
   - 2.1. Quick-Find
   - 2.2. Quick-Union
   - 2.3. Quick-Union Ponderado
3. Aplicaciones de UFDS

## Preguntas

# 1. Definición de UFDS

**¿Qué es UFDS?**

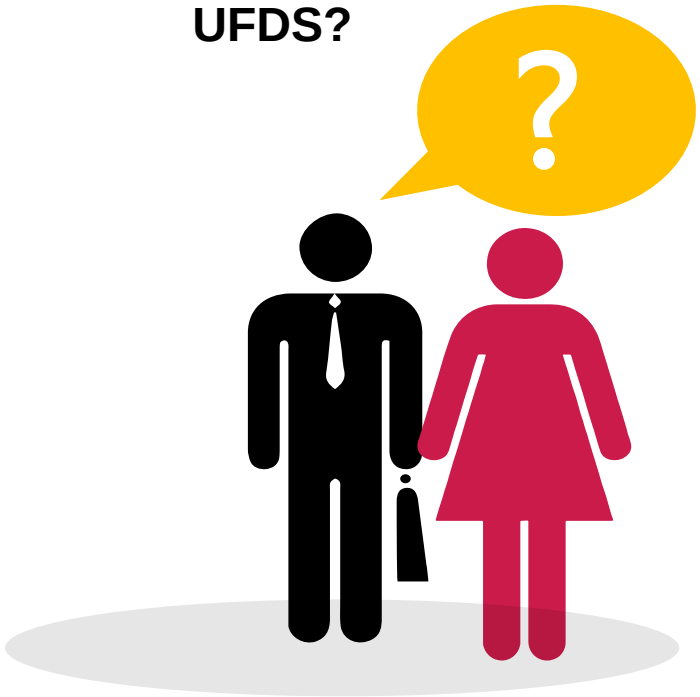

- Al hacer búsquedas y resolver problemas en grafos, hemos utilizado previamente técnicas como DFS o BFS. Ahora **UFDS** la utilizaremos para resolver un problema muy específico: **los componentes conectados**.
- No confundir con la detección de los componentes fuertemente conectados, que se resuelven con algoritmos DFS más complejos (algoritmo de Kosaraju).
- **Union-Find Disjoint Sets** (UFDS) por sus siglas en ingles, significa:

> **Estructura de datos de conjuntos disjuntos** es una estructura de datos que almacena una colección de conjuntos disjuntos (no superpuestos).

- En otras palabras, un conjunto disjunto es un grupo de conjuntos donde ningún elemento puede estar en más de un conjunto.

# 1. Definición de UFDS

**¿Para qué nos sirve UFDS?**




- Para identificar o rastrear elementos particionados en diferentes grupos o “sets”.
- Para detectar ciclos dentro de un grafo.
- Para mostrar conectividad

Para su implementación, veremos:

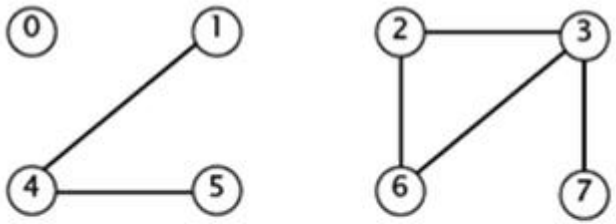
- Internamente utiliza un arreglo de padres.

# 1. Definición de UFDS

## UN POCO DE LO BASICO

Vamos a asumir que la conexión es una relación de equivalencia. Es decir:

- **Es reflexiva: p** está conectado con **p**
- **Es simétrica:** si **p** está conectado con **q**, **q** está conectado con **p**
- **Es transitiva:** si **p** está conectado con **q** y **q** está conectado con **r**, entonces **p** está conectado con **r**

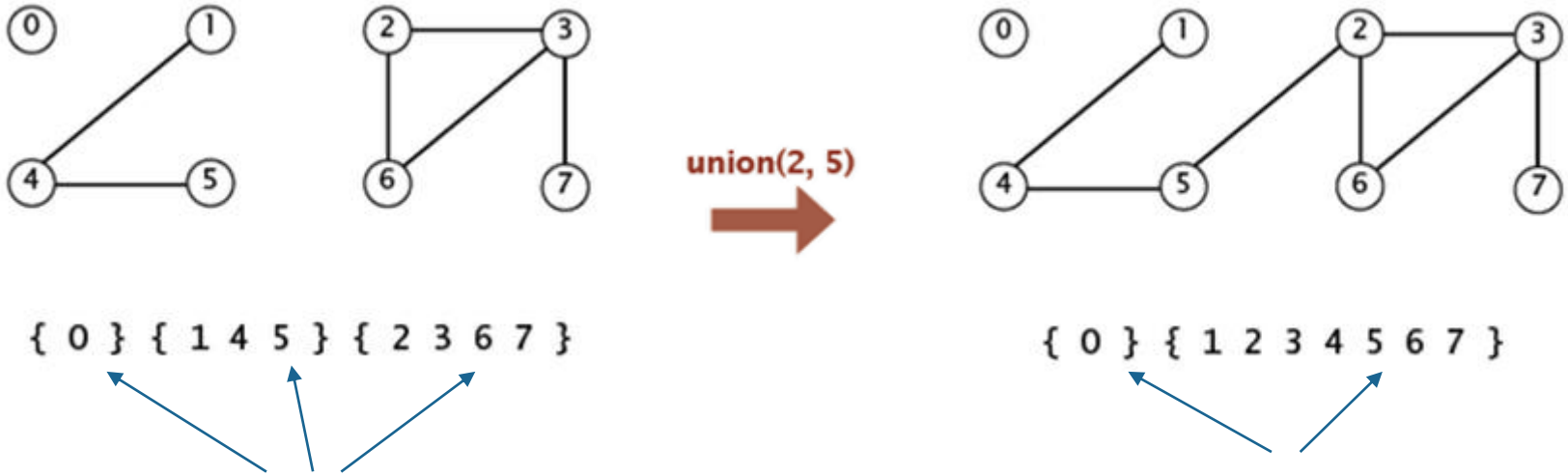



{ 0 } { 1 4 5 } { 2 3 6 7 }

3 componentes conexos

# 1. Definición de UFDS

**UN POCO DE LO BASICO**




3 componentes conexos

2 componentes conexos

# 2. Algoritmos o Estrategias Union-Find

- El **algoritmo de Union-Find**, también llamado algoritmo de búsqueda de unión, es un algoritmo que realiza dos operaciones útiles en esta estructura de datos:

1. **Buscar (Find):**
   - Determinar en qué subconjunto se encuentra un elemento en particular.
   - Esto se puede usar para determinar si dos elementos están en el mismo subconjunto.

2. **Unir (Union):**
   - Unir dos subconjuntos en un solo subconjunto.
   - Primero tenemos que verificar si los dos subconjuntos pertenecen al mismo conjunto. Si no, entonces no podemos realizar la unión.

# 2. Algoritmos o Estrategias Union-Find

**Ejemplo**

A partir de un Conjunto de objetos: 0 1 2 3 4 5 6 7 8 9

**Elementos particionados en conjuntos disjuntos (Disjoint Sets):**

0 1 {2 3 9} {5 6} 7 {4 8}

**Consultar para encontrar (Find):**

0 1 {**2** 3 **9**} {5 6} 7 {4 8}

Están los objetos 2 y 9 conectados?

**Comando de unión (Union):**

0 1 {2 3 4 8 9} {5 6} 7
Agregar una conexión entre los objetos 3 y 8.

# 2. Algoritmos o Estrategias Union-Find

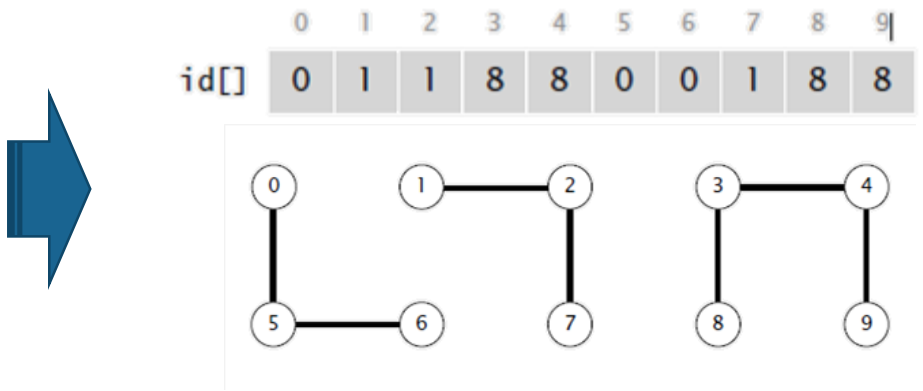
## 2.1. Algoritmo Quick-Find

**OBJETIVO:** El objetivo de este algoritmo es encontrar si dos elementos están conectados. Si no están conectados, los conectaremos.

- Este algoritmo, llamado también algoritmo entusiasta, para resolver el problema denominado: **problema de conectividad dinámica**.

**Consideraciones de Quick-Find:**

1. Arreglo id[] de tamaño N
2. Interpretación: **p** y **q** están conectados si tienen el mismo id.



# 2. Algoritmos o Estrategias Union-Find

## 2.1. Algoritmo Quick-Find

**Reglas de QUICK FIND:**

- La estructura de los datos de este algoritmo incluye:

  - Una matriz de enteros **id[]** de tamaño **N** (donde N es cualquier entero).

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | 9 | 9 | 9 | 6 | 6 | 7 | 8 | 9 |

  - Suponemos que la matriz de enteros **id[]** es un rango de 0 a N-1.

  - Los elementos **p** y **q** son 2 enteros en la matriz **id[]**.

  - Los elementos **p** y **q** están conectados si tienen el mismo id.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[i] | 0 | 1 | **9** | **9** | **9** | **6** | **6** | 7 | 8 | **9** |

**5** y **6** están conectados.

**2**, **3**, **4** y **9** están conectados.

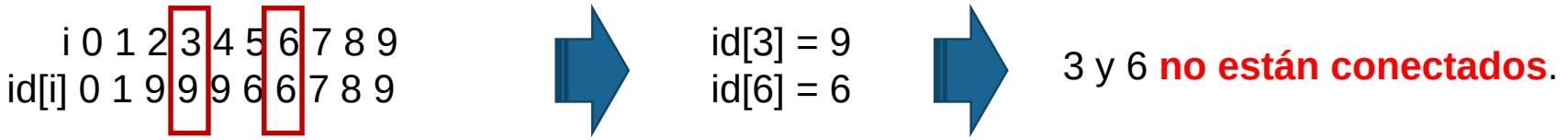
} Tienen el mismo id

# 2. Algoritmos o Estrategias Union-Find

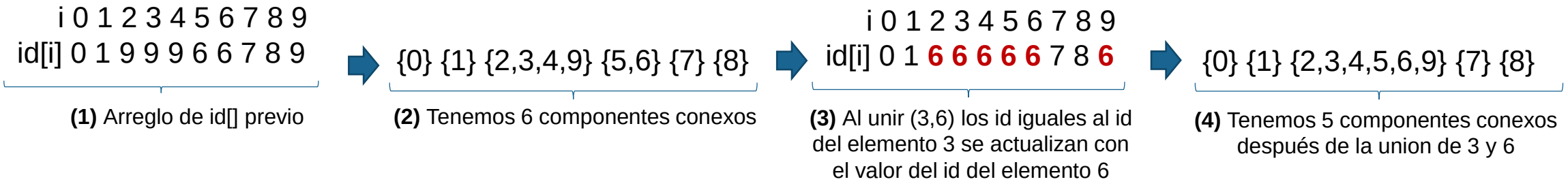
## 2.1. Algoritmo Quick-Find

**Ejemplo:**

**Paso 1 -> Find:** Verificar si **p** y **q** tienen el mismo id ⇨ ¿Los elementos 3 y 6 están conectados?




**Paso 2 -> Union:** Unir componentes conteniendo **p** y **q** ⇨ Unir 3 y 6




Los elementos 2, 3, 4, 5, 6 y 9 están ahora conectados.

# 2. Algoritmos o Estrategias Union-Find

## 2.1. Algoritmo Quick-Find

```
class QuickFind(object):
    def __init__(self, N):
        self.lst = list(range(N))


def find(self, p, q):
    return self.lst[p] == self.lst[q]


def union(self, p, q):
    pid = self.lst[p]

    qid = self.lst[q]

    for ind, x in enumerate(self.lst):
      if x == pid:
        self.lst[ind] = qid

    return self.lst
```

**Complejidad:**

- El algoritmo Quick-Find puede tomar M × N pasos para procesar M comandos de unión sobre N objetos.

**Ejemplo**

➥$10^{10}$ aristas conectando $10^{9}$ nodos.
➥Quick-Find tomará mas de $10^{19}$ operaciones.
➥300+ años de tiempo computacional.

| Algoritmo | Inicialización | Union | Consulta |
|---|---|---|---|
| QuickFind | N | N | 1 |

**Desventajas de QuickFind**

- Vemos que es muy lento en las uniones!
- Tenemos que agregar N elementos: $O(n^2)$

# 2. Algoritmos o Estrategias Union-Find

## 2.2. Algoritmo Quick-Union

- En el algoritmo de búsqueda rápida (Quick-Find), cada vez que hacíamos una unión, teníamos que iterar a través de toda la matriz. Eso no pasará en Quick-Union porque solo cambiaremos una identificación.

**OBJETIVO:** Mejorar el algoritmo de búsqueda rápida Quick-Find para que sea más eficiente.

- El enfoque principal estará en el método de 'unión'. Ese fue el método más ineficiente en Quick-Find
- Aquí ayudará un enfoque perezoso para el método de unión.

# 2. Algoritmos o Estrategias Union-Find

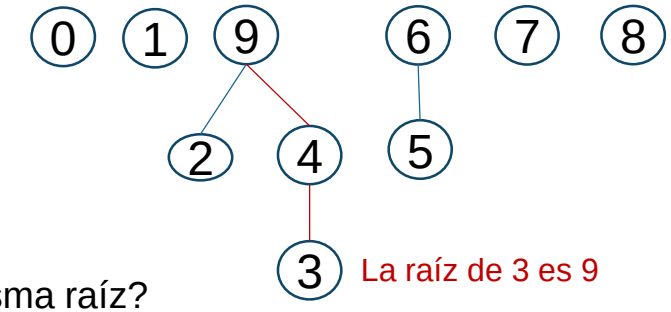
## 2.2. Algoritmo Quick-Union

### Reglas de QUICK UNION:

- Arreglo id[] de tamaño N
- Interpretación: **id[i]** es padre de **i**
- La **Raíz** de **i** es **id**[id[id[… id[i] …]]] (sigue hasta que no cambie, el algoritmo asegura que no haya ciclos)

### Interpretación:

i 0 1 2 3 4 5 6 7 8 9
**Raíz =** id[i] 0 1 9 4 9 6 6 7 8 9



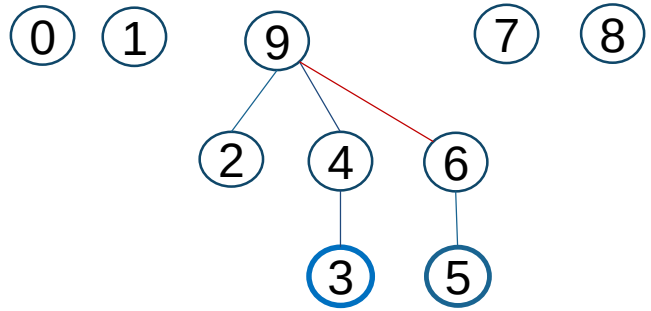

**Find:** p(3) y q(5) ¿Tienen la misma raíz?

**Union:** Unir componentes conteniendo p y q

Unir 3 y 5 = Unir **Raíz** de 3 y 5 = Unir 9 y 6

i 0 1 2 3 4 5 6 7 8 9
**Raíz =** id[i] 0 1 9 4 9 6 **9** 7 8 9

**Solo un valor cambia**



# 2. Algoritmos o Estrategias Union-Find

## 2.2. Algoritmo Quick-Union

```
class QuickUnion(object):
    def __init__(self, N):
      self.lst = list(range(N))

    def find(self, ind):
      while ind != self.lst[ind]:
        ind = self.lst[ind]
      return ind

    def connect(self, p, q):
      return self.find(p) == self.find(q)

    def union(self, p, q):
      pid = self.find(p)
      self.lst[pid] = self.find(q)
```

**Complejidad:**

- La unión puede ser muy costosa (N pasos).
- Los arboles pueden volverse muy altos Find puede ser también muy costoso (N pasos).
- Necesario hacer Find para hacer Union.

| Algoritmo | Inicialización | Union | Consulta |
|---|---|---|---|
| QuickUnion | N | N+ | N |

**Desventajas de QuickUnion:**

- Los árboles podrían quedar muy grandes
- La consulta puede ser muy cara

# 2. Algoritmos o Estrategias Union-Find

## 2.3. Algoritmo Quick-Union Ponderado

**OBJETIVOS:**

- Modificar el algoritmo Quick-Union para evitar arboles altos.
- Mantener registro del tamaño de cada componente.
- Balancear conectando los arboles pequeños debajo del más grande.

Se incorporan dos tipos de mejoras:

**Mejora #1:** **Árbol Ponderado**

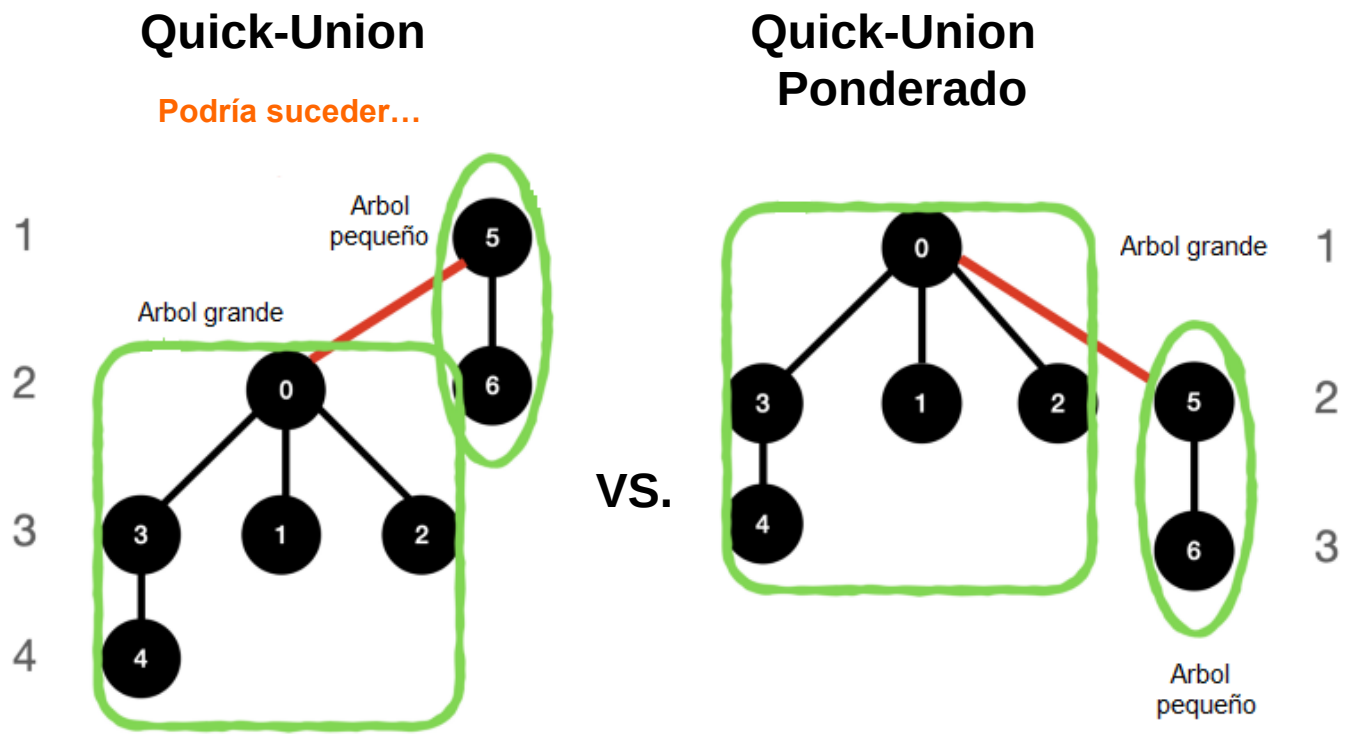
**Mejora #2:** Compresión de la ruta

# 2. Algoritmos o Estrategias Union-Find

## 2.3. Algoritmo Quick-Union Ponderado

**Mejora #1: Árbol Ponderado**

- La idea básica del árbol ponderado es que siempre coloca el árbol más pequeño (el árbol con menos nodos) debajo del árbol más grande (el árbol con más nodos).
- En **Quick-Union**, cuando vinculamos dos árboles de diferentes tamaños, podemos vincular el árbol más grande debajo del árbol más pequeño, creando más capas.
- En **Quick-Union Ponderado**, examinaremos el tamaño de dos árboles y nos aseguraremos de unir solo el árbol más pequeño debajo del árbol más grande.
- Al aplicar este método, podemos garantizar que el árbol no crecerá demasiado alto.

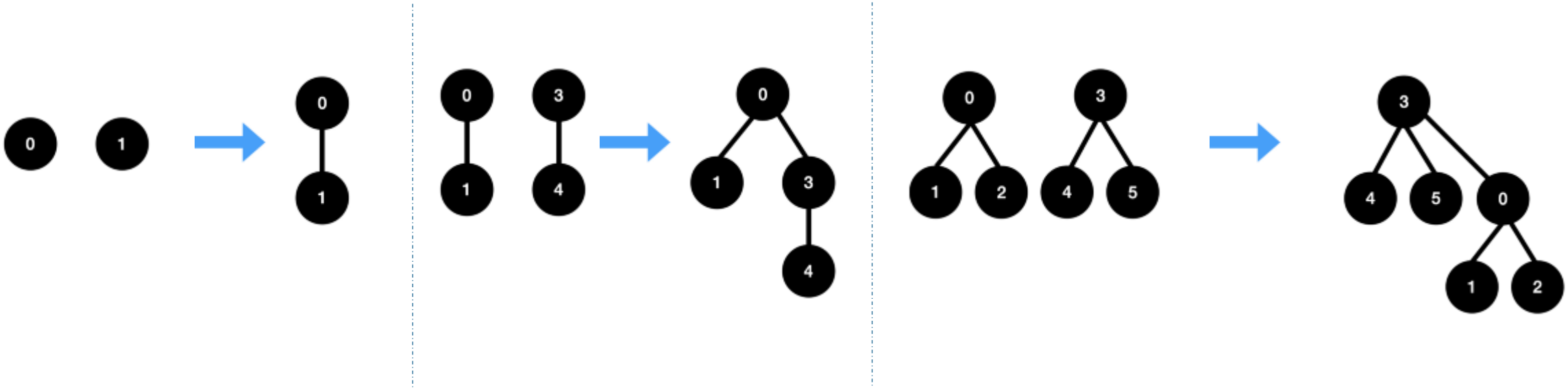


# 2. Algoritmos o Estrategias Union-Find

## 2.3. Algoritmo Quick-Union Ponderado

**Mejora #1: Árbol Ponderado**

- La estructura de árbol de **Quick-Union Ponderado** aumentará una capa más cuando el número de dos capas de árbol sea el mismo .



# 2. Algoritmos o Estrategias Union-Find

## 2.3. Algoritmo Quick-Union Ponderado

```
def find(s, a):
    i = a
    while s[i] >= 0:
        i = s[i]
    return i

def union(s, a, b):
    pa = find(s, a)
    pb = find(s, b)
    if pa == pb: return
    if s[pa] < s[pb]:
        s[pa] += s[pb]
        s[pb] = pa
    elif s[pb] < s[pa]:
        s[pb] += s[pa]
        s[pa] = pb
    else:
        s[pa] += s[pb]
        s[pb] = pa
```

**Mejora en la función Union**

**Find:** Si **p** y **q** están conectados no hacemos nada. Si no lo están los unimos.

**Union:** Al unir componentes, revisaremos su ponderación en cuanto a tamaño, recalculando la ponderación y asignando un valor nuevo de raíz según los siguientes casos:

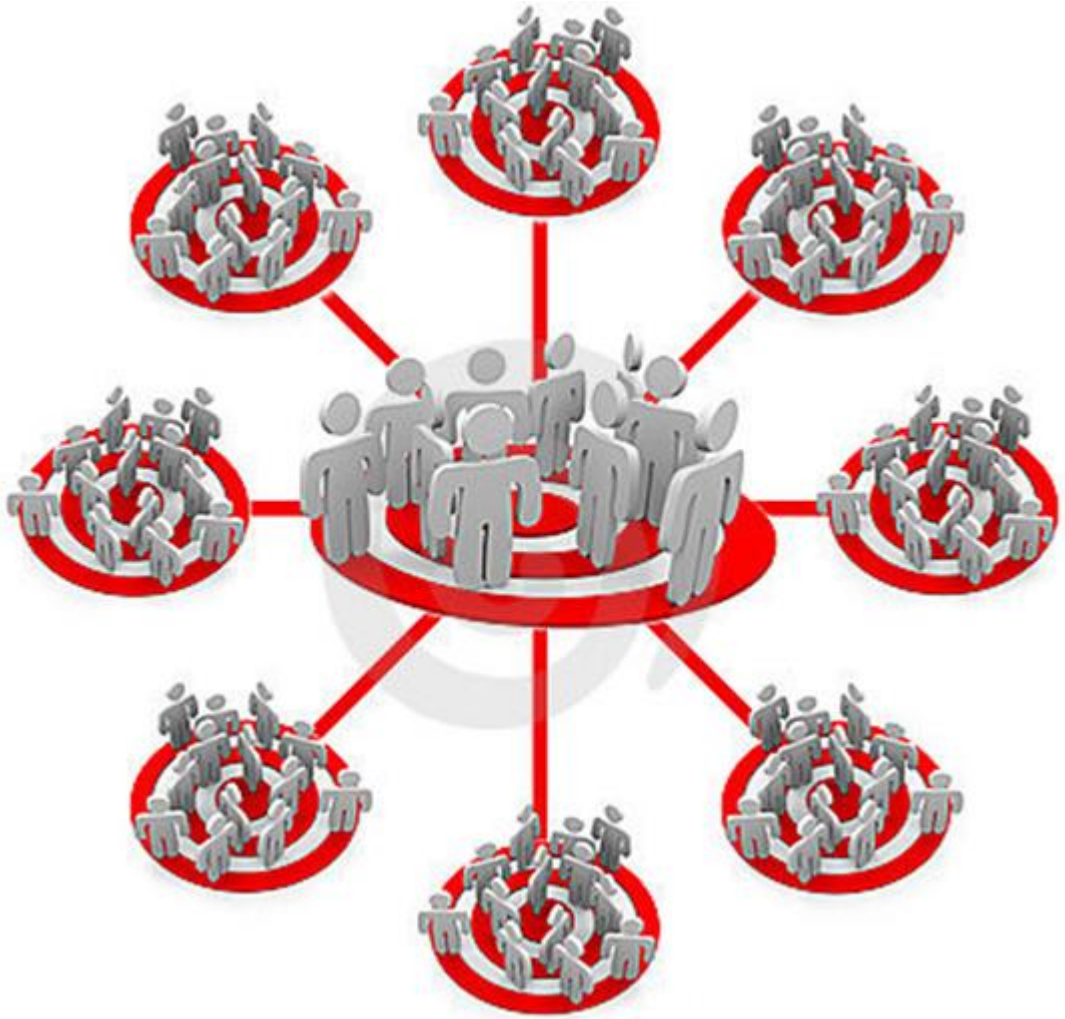
- **Ponderación de p < ponderación de q**, recalculamos ponderación para p y actualizamos la raíz de q con la de p.
- **Ponderación de p > ponderación de q**, recalculamos ponderación para q y actualizamos la raíz de p con la de q.
- **Igual ponderación de p y q**, recalculamos ponderación para p y actualizamos la raíz de q con la de p (igual que en el primer caso).

**Complejidad**

| Algoritmo | Inicialización | Union | Consulta |
|---|---|---|---|
| QuickUnion Ponderado | N | Lg *N* | Lg *N* |

- **Find:** toma un tiempo proporcional a la profundidad de **p** y **q**.
- **Union:** toma un tiempo constante, dadas las raíces.
- La profundidad es a lo sumo de log N

# 3. Aplicaciones de UFDS



**Manejar grupos eficientemente**

# 3. Aplicaciones de UFDS

Estos tipos de algoritmos ayudan a manipular los objetos de todo tipo:

- Computadoras en una red.
- Elementos en un conjunto matemático.
- Sitios metálicos en un sistema compuesto.
- Píxeles en una foto digital.
- Amigos en una red social.
- Transistores en un chip de computadora.

# PREGUNTAS

Dudas y opiniones