# Hoja de Ejercicios 9
# Complejidad Algorítmica - UPC

Considere una situación con un número de personas y siguiendo las tareas a realizar en ellas.
Agregar una nueva relación de amistad, es decir, una persona x se convierte en amiga de otra persona y.
Encuentra si el individuo x es un amigo del individuo y (amigo directo o indirecto)

**Ejemplo**

- Nos dan 10 individuos: a, b, c, d, e, f, g, h, i, j
- Las siguientes son las relaciones que deben agregarse.
  a $\iff$ b
  b $\iff$ d
  c $\iff$ f c
  $\iff$ i j
  $\iff$ e g
  $\iff$ j
- Y tenemos consultas como: a es amigo de d o no
- Básicamente necesitamos crear los siguientes 4 grupos: G1 = {a, b, d}
  G2 = {c, f, i}
  G3 = {e, g, j}
  G4 = {h}
- Mantener una conexión de acceso rápido entre los elementos de los grupos.

Problema: encontrar si x e y pertenecen al mismo grupo o no, es decir, para encontrar si x e y son amigos directos / indirectos.

Solución: dividir a los individuos en diferentes conjuntos de acuerdo con los grupos en los que caen. Este método se conoce como estructura de datos de conjuntos disjuntos (UFDS) que mantiene la colección de conjuntos disjuntos y cada conjunto está representado por su representante, que es uno de sus miembros.

- Enfoque:
  - ¿Cómo resolver conjuntos? Inicialmente todos los elementos pertenecen a diferentes conjuntos. Después de trabajar en las relaciones dadas, seleccionamos un miembro como representante. Puede haber muchas formas de seleccionar un representante, una simple es seleccionar con el índice más grande.
  - ¿Comprobar si 2 personas están en el mismo grupo? Si los representantes de dos personas son iguales, entonces se convertirán en amigos.
- Estructuras de datos usados:
  - Array: un arreglo de enteros, llamado parent [ ]. Si estamos tratando con n elementos, el i-ésimo elemento del arreglo es el padre del i-ésimo elemento.
  - Árbol: Es un conjunto desunido. Si dos elementos están en el mismo árbol, entonces están en el mismo conjunto disjunto. El nodo raíz (o nodo superior) de cada árbol se denomina como

el representante del conjunto. Siempre hay un único representante único de cada conjunto. Una regla simple para identificar al representante es, si i es el representante de un conjunto, entonces padre[i] = i.

- Operaciones:
  - Find: Puede implementarse atravesando recursivamente el arreglo parent [] hasta que lleguemos a un nodo que sea padre de sí mismo.

```
//Encontrar el representante del conjunto conteniendo i
public int find(int i) {
    // Si i es su propio padre
    if (parent[i] == i) {
        // Entonces i es el representante del conjunto
        return i;
    }
    else {
        // Sino i no es el representante de este conjunto.
        // Por ende, se llama recursivamente Find sobre su padre
        return find(parent[i]);
    }
}
```

  - Union: Toma, como entrada, dos elementos. Y encuentra a los representantes de sus conjuntos utilizando la operación Find, y finalmente coloca uno de los árboles (que representa el conjunto) debajo del nodo raíz del otro árbol, fusionando efectivamente los árboles y los conjuntos.

```
// Unir los conjuntos que contienen i y j respectivamente
public void union(int i, int j) {
    // Encontrar el representante del conjunto que contiene a i
    int irep = this.Find(i);
    // Hacer lo mismo para el conjunto que contiene j
    int jrep = this.Find(j);
    // Hacer que el padre del representante de i sea el
       representante de j.
    // Efectivamente, moviendo todo el conjunto conteniendo i
       en el conjunto conteniendo j
    this.Parent[irep] = jrep;
}
```

## Ejercicio 1: UFDS

Resolver este problema requiere implementar las funciones básicas y probar su complejidad sobre casos prácticos.

### Ejercicio 1.1

Implemente las funciones Find y Union.

### Ejercicio 1.2

Implemente un algoritmo para gestionar el grupo de amigos descrito en la primera parte.

# Hoja de Ejercicios 9
# Complejidad Algorítmica - UPC

## Ejercicio 2: Mejoras

La eficiencia depende en gran medida de la altura del árbol. Necesitamos minimizar la altura del árbol para mejorar la eficiencia. Podemos usar los métodos de compresión de caminos y unión por rangos para hacerlo.

### Ejercicio 2.1: Compresión de caminos

Para acelerar la estructura de datos al comprimir la altura de los árboles. Se puede lograr insertando un pequeño mecanismo de almacenamiento en la operación Find.

### Ejercicio 2.2: Unión por rangos

En primer lugar, necesitamos un nuevo arreglo de enteros llamado rank[]. El tamaño de este arreglo es el mismo que el del arreglo parent[]. Si i es un representante de un conjunto, el `rank[i]` es la altura del árbol que representa el conjunto. Ahora recuerde que, en la operación de Unión, no importa cuál de los dos árboles se mueve debajo del otro. Ahora lo que queremos hacer es minimizar la altura del árbol resultante. Si estamos uniendo dos árboles (o conjuntos), llamémoslos izquierda y derecha, entonces todo depende del rango de izquierda y del rango de derecha.

- Si el rango de la izquierda es menor que el de la derecha, entonces es mejor moverse a la izquierda debajo de la derecha.
- De la misma manera, si el rango de la derecha es menor que el de la izquierda, entonces deberíamos movernos a la derecha debajo de la izquierda.
- Si los rangos son iguales, no importa qué árbol vaya debajo del otro, pero el rango del resultado siempre será uno mayor que el rango de los árboles.

## Ejercicio 3: Detección de ciclos en grafos no dirigidos

Implemente un método basado en UFDS para detectar ciclos en grafos no dirigidos

# Hoja de Ejercicios 10
# Complejidad Algorítmica - UPC

## Ejercicio 1: MST

Implemente un algoritmo de fuerza bruta para obtener el MST de un grafo.

## Ejercicio 2: Prim

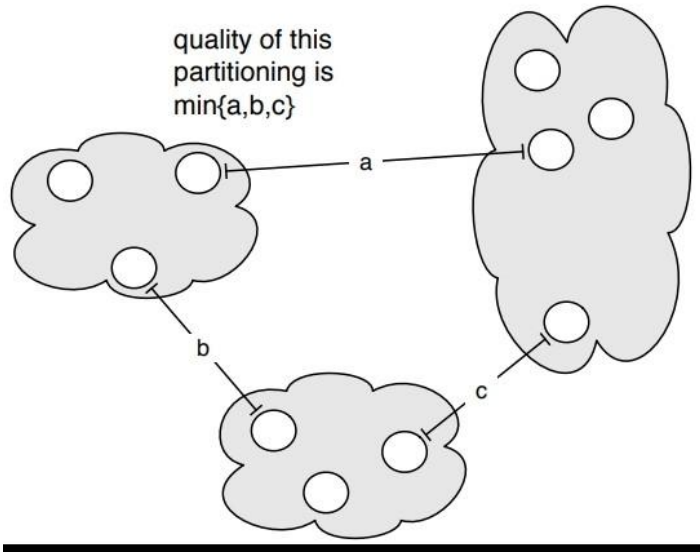
Implemente el algoritmo de Prim para obtener el MST de un grafo.

## Ejercicio 3: Kruskal

Implemente el algoritmo de Kruskal para obtener el MST de un grafo.

## Ejercicio 4: Clustering

A Ud. se le entregan n objetos y la distancia d (u, v) entre cada par de objetos.
d (u, v) puede ser una distancia real, o alguna representación abstracta de cuan similares son (e.g. ¿cuál es la "distancia" entre 2 especies de animales?).
Objetivo: Dividir los n objetos en k grupos tal que la distancia mínima entre objetos en diferentes grupos sea maximizada.



# Hoja de Ejercicios 10
# Complejidad Algorítmica - UPC

## Ejercicio 1: MST

Implemente un algoritmo de fuerza bruta para obtener el MST de un grafo.

## Ejercicio 2: Prim

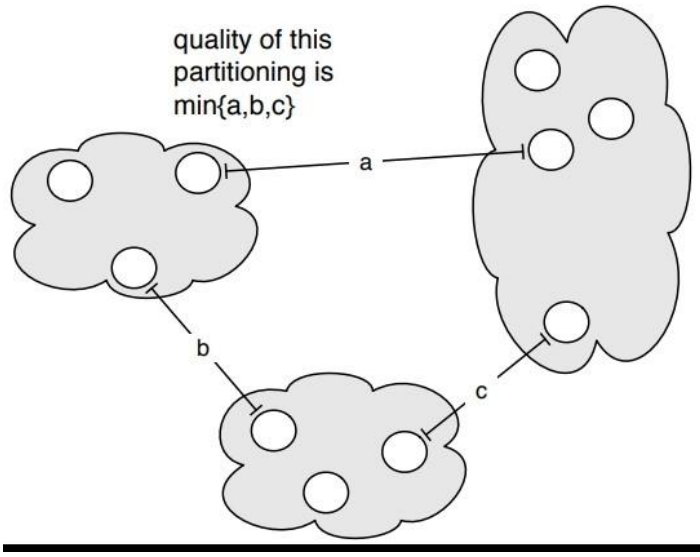
Implemente el algoritmo de Prim para obtener el MST de un grafo.

## Ejercicio 3: Kruskal

Implemente el algoritmo de Kruskal para obtener el MST de un grafo.

## Ejercicio 4: Clustering

A Ud. se le entregan n objetos y la distancia d (u, v) entre cada par de objetos.
d (u, v) puede ser una distancia real, o alguna representación abstracta de cuan similares son (e.g. ¿cuál es la "distancia" entre 2 especies de animales?).
Objetivo: Dividir los n objetos en k grupos tal que la distancia mínima entre objetos en diferentes grupos sea maximizada.



# Hoja de Ejercicios 12
# Complejidad Algorítmica - UPC

## Ejercicio 1: Fibonacci

Implemente un algoritmo de programación dinámica para calcular la sucesión de Fibonacci.

## Ejercicio 2: El problema de la diligencia

Una diligencia debe atravesar el oeste estadounidense en plena fiebre del oro. Cada uno de los tramos de su recorrido está cubierto por una póliza de seguro, cuyo costo es directamente proporcional al riesgo presente durante el viaje. El recorrido se inicia en la ciudad A y tiene como destino la ciudad J. La figura 1 ilustra la situación. Los números en los arcos indican el costo de la póliza que cubre el viaje entre las dos ciudades, por ejemplo, la póliza del viaje entre A y C tiene un costo de 3. El conductor supone que la ruta más segura es aquella para la cual la suma total de los costos de las pólizas sea mínima.

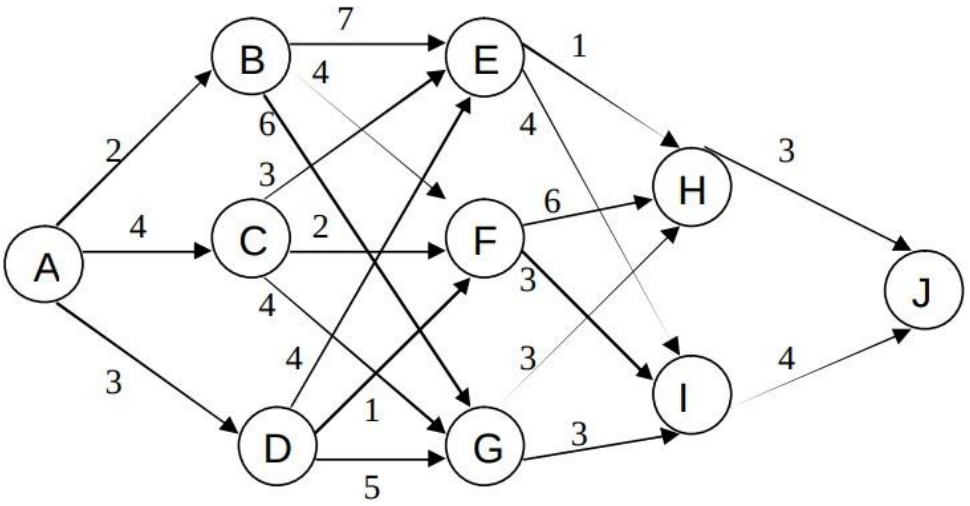



Figura 1: Red de vías.

## Ejercicio 3: La función de Ackermann

La función de Ackermann se define recursivamente del modo siguiente:

$$\begin{cases} Ack(0,n) = n+1 \\ Ack(m,0) = Ack(m-1,1)\ si\ m > 0 \\ Ack(m,n) = Ack(m-1, Ack(m,n-1))\ si\ m,n > 0. \end{cases}$$

Implementar esta función con programación dinámica.

# Hoja de Ejercicios 12
# Complejidad Algorítmica - UPC

---

## Ejercicio 4: Intereses bancarios

Dadas n funciones $f_1, f_2, ..., f_n$ y un entero positivo M, deseamos maximizar la función $f_1(x_1) + f_2(x_2) + ... + f_n(x_n)$ sujeta a la restricción $x1 + x2 + ... + xn = M$, donde $fi(0) = 0$ $(i = 1, ..., n)$, $x_i$ son números naturales, y todas las funciones son monótonas crecientes, es decir, $x \geq y$ implica que $fi(x) > fi(y)$. Supóngase que los valores de cada función se almacenan en un vector. Este problema tiene una aplicación real muy interesante, en donde $f_i$ representa la función de interés que proporciona el banco i, y lo que deseamos es maximizar el interés total al invertir una cantidad determinada de dinero M. Los valores $x_i$ van a representar la cantidad a invertir en cada uno de los n bancos.

# Hoja de Ejercicios 13
# Complejidad Algorítmica - UPC

---

## Ejercicio 1: Bellman - Ford

Implemente el algoritmo de Bellman - Ford.

## Ejercicio 2: Floyd - Warshall

Implemente el algoritmo de Floyd - Warshall.

## Ejercicio 3: Ciclos negativos

Implemente un algoritmo que detecte si un grafo contiene un ciclo negativo (Un ciclo negativo es aquel en el que la suma total del ciclo es negativa).