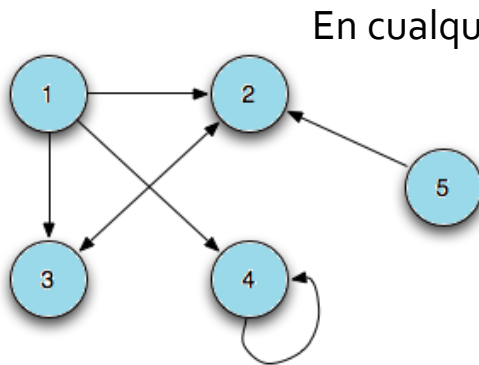


# Grafos en JAVA

## Terminología

Un grafo es un conjunto de nodos que mantienen relaciones entre ellos. Podemos definir a un grafo como un par de conjuntos finitos  $G=(V,A)$  donde  $V$  es el conjunto finito de vértices y  $A$  es el conjunto finito de Aristas.

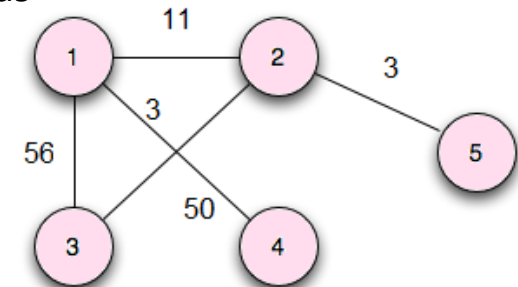
Por lo general, los **vértices** son nodos de procesamiento o estructuras que contienen algún tipo de información mientras que las **aristas** determinan las relaciones entre nodos. Las aristas también pueden tener algún tipo de información asociada (distancia, peso, etc.) en cuyo caso es un **grafo pesado**.



En cualquiera de los dos grafos, las relaciones las determinan las **aristas**.

Si se utilizan flechas para conectar los nodos decimos que el **grafo es dirigido** (también llamado **digrafo**).

Grafo no pesado



Si la conexión entre los vértices no tiene dirección, el **grafo es no dirigido**.

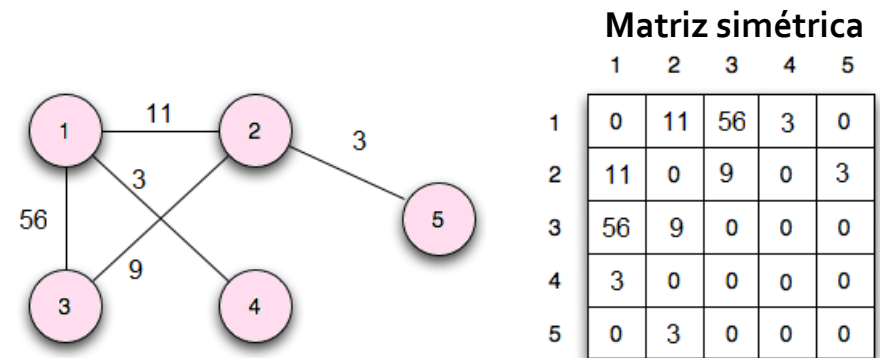
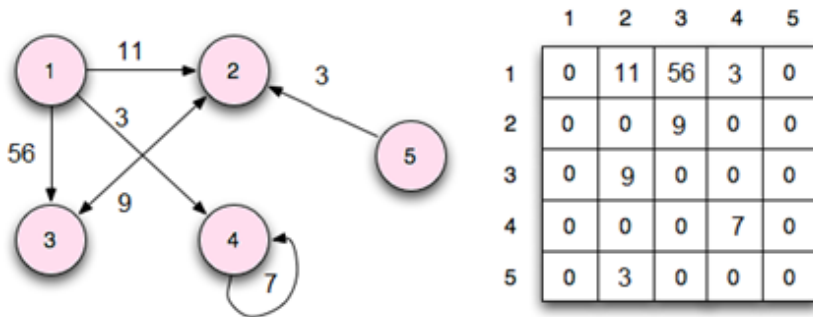
Grafo pesado

# Grafos en JAVA

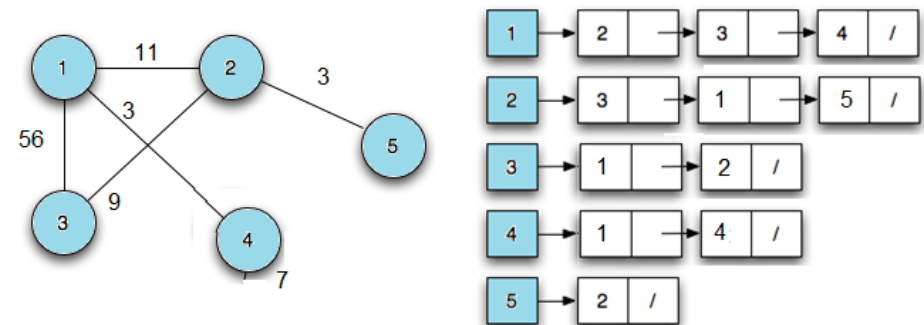
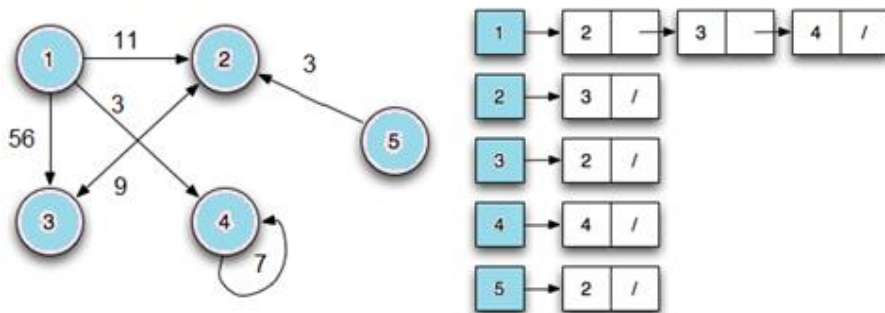
## Representaciones

Los métodos mas comunes para representar grafos son matriz de adyacencias y lista de adyacencias.

- **Matiz de adyacencias:** el grafo se representa como una matriz de  $|V| \times |V|$ , con valores enteros (o de otro tipo de dato).



- **Lista de adyacencias:** el grafo  $G=(V,A)$  se representa como un arreglo/lista de  $|V|$  de vértices.



# Grafos en JAVA

## La interfaces para definir Grafos

<<Java Interface>>

**I Grafo<T>**

tp09.ejercicio4

- agregarVertice(Vertice<T>):void
- eliminarVertice(Vertice<T>):void
- conectar(Vertice<T>,Vertice<T>):void
- conectar(Vertice<T>,Vertice<T>,int):void
- desConectar(Vertice<T>,Vertice<T>):void
- esAdyacente(Vertice<T>,Vertice<T>):boolean
- listaDeVertices():ListaGenerica<Vertice<T>>
- listaDeAdyacentes(Vertice<T>):ListaGenerica<Arista<T>>
- getVertice(int):Vertice<T>
- getPeso(Vertice<T>,Vertice<T>):int
- esVacio():boolean

<<Java Interface>>

**I Arista<T>**

tp09.ejercicio4

- getVerticeDestino():Vertice<T>
- getPeso():int

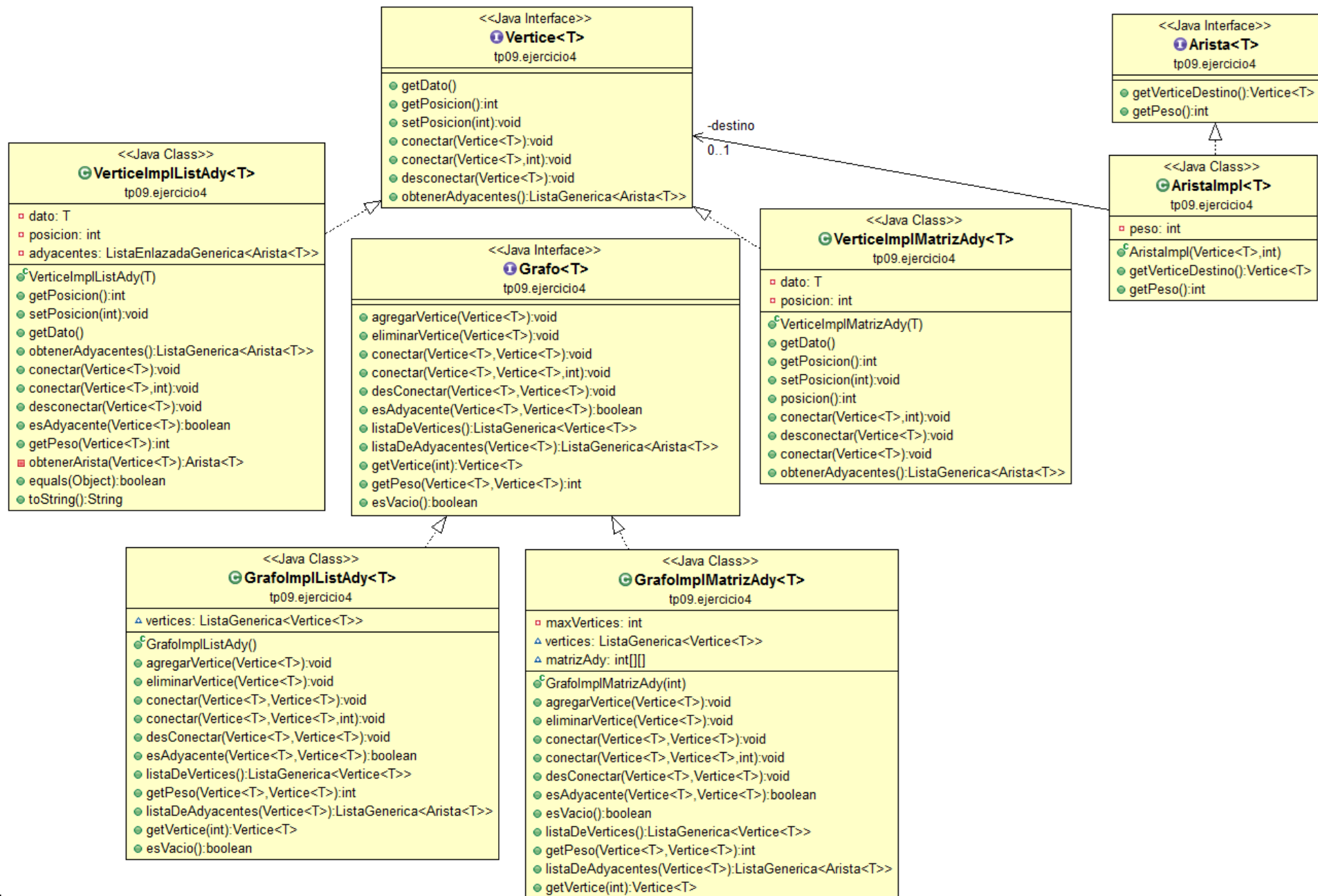
<<Java Interface>>

**I Vertice<T>**

tp09.ejercicio4

- getDato()
- getPosicion():int
- setPosicion(int):void
- conectar(Vertice<T>):void
- conectar(Vertice<T>,int):void
- desconectar(Vertice<T>):void
- obtenerAdyacentes():ListaGenerica<Arista<T>>

# Grafos - La interfaces y clases



# Grafos en JAVA

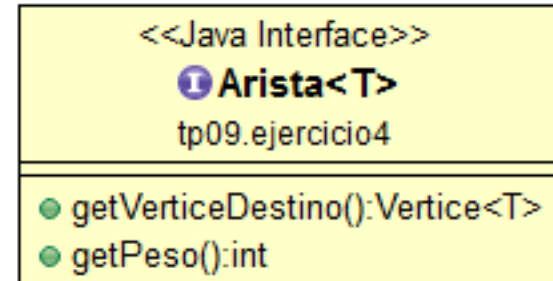
## La clase que implementa a la interface Arista

```
public class AristaImpl<T> implements Arista<T> {
    private Vertice<T> destino;
    private int peso;

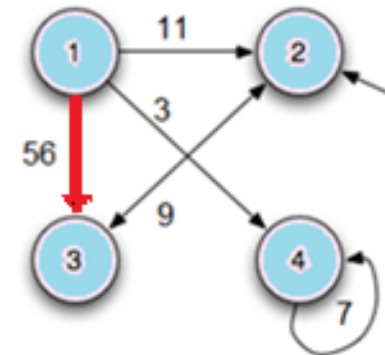
    public AristaImpl(Vertice<T> dest, int p){
        destino = dest;
        peso = p;
    }

    @Override
    public Vertice<T> getVerticeDestino() {
        return destino;
    }

    @Override
    public int getPeso() {
        return peso;
    }
}
```



**Arista:** una arista siempre tiene el destino y podría tener un peso.



# Grafos con Lista de Adyacencias

## Clase que implementa la interface Vertice

```
public class VerticeImplListAdy<T> implements Vertice<T> {
    private T dato;
    private int posicion;
    private ListaEnlazadaGenerica<Arista<T>> adyacentes;

    public VerticeImplListAdy(T d) {
        dato = d;
        adyacentes = new ListaEnlazadaGenerica<Arista<T>>();
    }

    public ListaGenerica<Arista<T>> obtenerAdyacentes() {
        return adyacentes;
    }

    public void conectar(Vertice<T> v) {
        conectar(v, 1);
    }

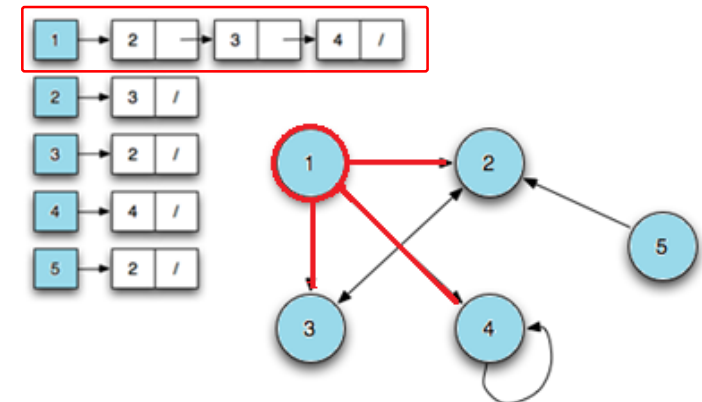
    public void conectar(Vertice<T> v, int peso) {
        Arista<T> a = new AristaImpl<T>(v, peso);
        if (!adyacentes.incluye(a))
            adyacentes.agregarInicio(a);
    }

    public void desconectar(Vertice<T> v) {
        Arista<T> aristaAborrar = obtenerArista(v);
        adyacentes.eliminar(aristaAborrar);
    }

    ...
}
```

<<Java Interface>>	
① Vertice<T>	
tp09.ejercicio4	
•	getDato()
•	getPosicion():int
•	setPosicion(int):void
•	conectar(Vertice<T>):void
•	conectar(Vertice<T>,int):void
•	desconectar(Vertice<T>):void
•	obtenerAdyacentes():ListaGenerica<Arista<T>>

**Vértice:** tiene un dato y una lista de adyacentes. La lista es de aristas, donde cada una tiene un vértice destino.





# Grafos con Lista de Adyacencias

## Clase que implementa la interface Grafo

```
public class GrafoImplListAdy<T> implements Grafo<T> {
    ListaGenerica<Vertice<T>> vertices = new ListaEnlazadaGenerica<Vertice<T>>();

    @Override
    public void agregarVertice(Vertice<T> v) {
        if (!vertices.incluye(v)) {
            vertices.agregarFinal(v);
            v.setPosicion(vertices.tamano());
        }
    }

    @Override
    public void conectar(Vertice<T> origen, Vertice<T> destino) {
        origen.conectar(destino);
    }

    @Override
    public void conectar(Vertice<T> origen, Vertice<T> destino, int peso) {
        origen.conectar(destino, peso);
    }

    @Override
    public void desConectar(Vertice<T> origen, Vertice<T> destino) {
        origen.desconectar(destino);
    }

    @Override
    public boolean esAdyacente(Vertice<T> origen, Vertice<T> destino) {
        return origen.esAdyacente(destino);
    }

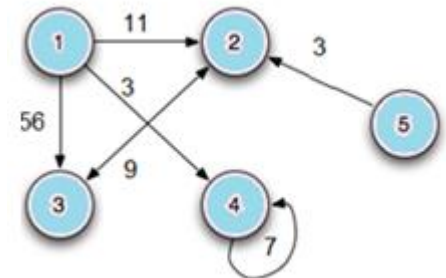
    @Override
    public ListaGenerica<Vertice<T>> listaDeVertices() {
        return vertices;
    }
    ...
}
```

<<Java Interface>>

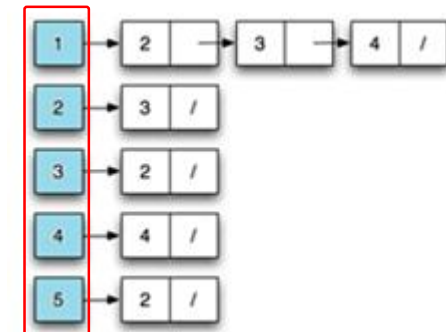
**Grafo<T>**

tp09.ejercicio4

- agregarVertice(Vertice<T>):void
- eliminarVertice(Vertice<T>):void
- conectar(Vertice<T>,Vertice<T>):void
- conectar(Vertice<T>,Vertice<T>,int):void
- desConectar(Vertice<T>,Vertice<T>):void
- esAdyacente(Vertice<T>,Vertice<T>):boolean
- listaDeVertices():ListaGenerica<Vertice<T>>
- listaDeAdyacentes(Vertice<T>):ListaGenerica<Arista<T>>
- getVertice(int):Vertice<T>
- getPeso(Vertice<T>,Vertice<T>):int
- esVacio():boolean



**Grafo:** tiene una lista de Vertices



# Grafos con Matriz de Adyacencias

## Clase que implementa la interface Vertice

```
public class VerticeImplMatrizAdy<T> implements Vertice<T> {
    private T dato;
    private int posicion;

    public VerticeImplMatrizAdy(T d) {
        dato = d;
    }

    @Override
    public T getDato() {
        return this.dato;
    }

    public int getPosicion() {
        return posicion;
    }

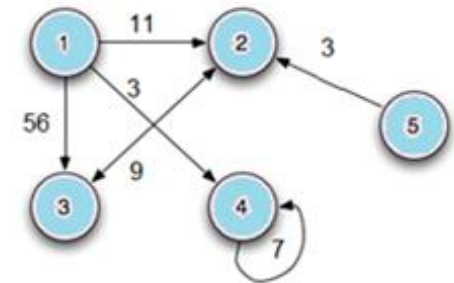
    public void setPosicion(int posicion) {
        this.posicion = posicion;
    }

    public int posicion(){
        return posicion;
    }

    @Override
    public void conectar(Vertice<T> v, int peso) {
    }

    @Override
    public void desconectar(Vertice<T> v) {
    }
    ...
}
```

<<Java Interface>>	
① <b>Vertice&lt;T&gt;</b>	
tp09.ejercicio4	
•	getDato()
•	getPosicion():int
•	setPosicion(int):void
•	conectar(Vertice<T>):void
•	conectar(Vertice<T>,int):void
•	desconectar(Vertice<T>):void
•	obtenerAdyacentes():ListaGenerica<Arista<T>>



**Vértice:** tiene un dato y la posición dentro de la lista.

	1	2	3	4	5
1	0	11	56	3	0
2	0	0	9	0	0
3	0	9	0	0	0
4	0	0	0	7	0
5	0	3	0	0	0



# Grafos con Matriz de Adyacencias

## Clase que implementa la interface Grafo

```
public class GrafoImplMatrizAdy<T> implements Grafo<T> {
    private int maxVertices;
    ListaGenerica<Vertice<T>> vertices;
    int[][] matrizAdy;

    public GrafoImplMatrizAdy(int maxVert){
        maxVertices = maxVert;
        vertices = new ListaEnlazadaGenerica<Vertice<T>>();
        matrizAdy = new int[maxVertices][maxVertices];
    }

    @Override
    public void agregarVertice(Vertice<T> v) {
        if (!vertices.incluye(v)){
            vertices.agregarFinal(v);
            v.setPosicion(vertices.tamano());
        }
    }

    @Override
    public void conectar(Vertice<T> origen, Vertice<T> destino) {
        conectar(origen, destino, 1);
    }

    @Override
    public void conectar(Vertice<T> origen, Vertice<T> destino, int peso) {
        matrizAdy[origen.getPosicion()][destino.getPosicion()] = peso;
    }

    @Override
    public void desConectar(Vertice<T> origen, Vertice<T> destino) {
        matrizAdy[origen.getPosicion()][destino.getPosicion()] = 0;
    }
    ...
}
```

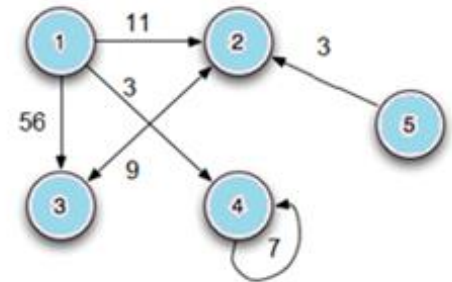
<<Java Interface>>

**Grafo<T>**

tp09.ejercicio4

- agregarVertice(Vertice<T>):void
- eliminarVertice(Vertice<T>):void
- conectar(Vertice<T>,Vertice<T>):void
- conectar(Vertice<T>,Vertice<T>,int):void
- desConectar(Vertice<T>,Vertice<T>):void
- esAdyacente(Vertice<T>,Vertice<T>):boolean
- listaDeVertices():ListaGenerica<Vertice<T>>
- listaDeAdyacentes(Vertice<T>):ListaGenerica<Arista<T>>
- getVertice(int):Vertice<T>
- getPeso(Vertice<T>,Vertice<T>):int
- esVacio():boolean

**Grafo: tiene una lista de Vertice y la matriz**



	1	2	3	4	5
1	0	11	56	3	0
2	0	0	9	0	0
3	0	9	0	0	0
4	0	0	0	7	0
5	0	3	0	0	0

# Agenda - Grafos

## Recorridos

- en profundidad: **DFS** (Depth First Search)
- en amplitud: **BFS** (Breath First Search)

# Grafos

## DFS (Depth First Search)

El DFS es un algoritmo de recorrido de grafos en profundidad. Generalización del recorrido preorden de un árbol.

### Esquema recursivo

Dado  $G = (V, A)$

1. Marcar todos los vértices como no visitados.
2. Elegir vértice  $u$  (no visitado) como punto de partida.
3. Marcar  $u$  como visitado.
4. Para todo  $v$  adyacente a  $u$ ,  $(u,v) \in A$ , si  $v$  no ha sido visitado, repetir recursivamente (3) y (4) para  $v$ .

### ¿Cuándo finaliza el recorrido?

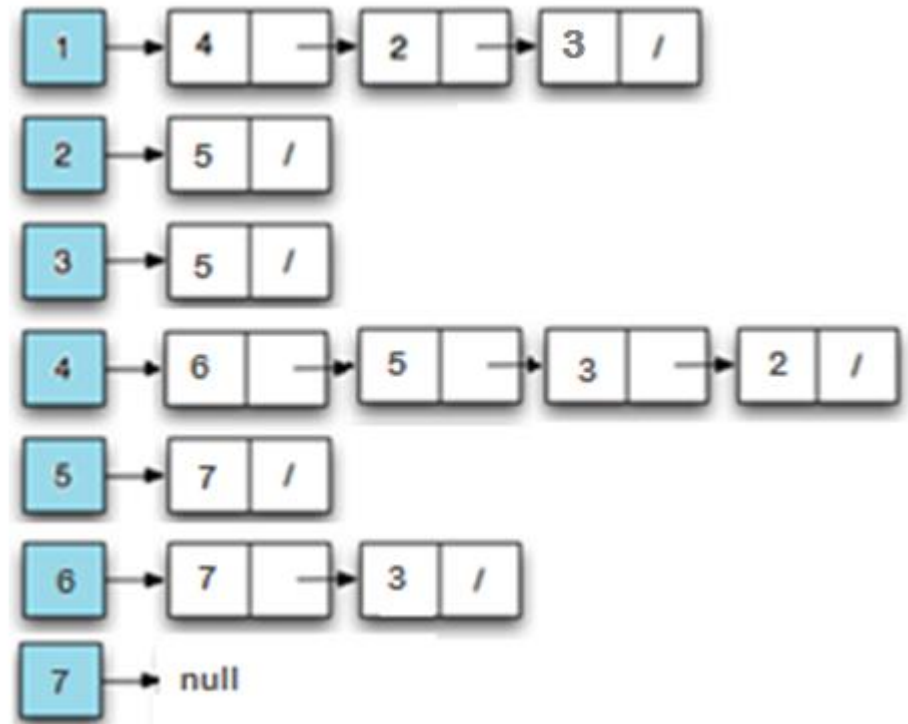
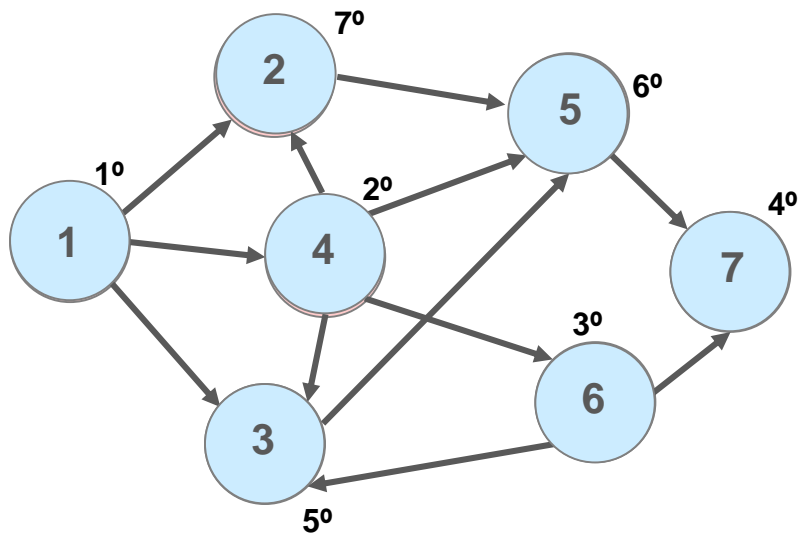
Finalizar cuando se hayan visitado todos los nodos alcanzables desde  $u$ .

Si desde  $u$  no fueran alcanzables todos los nodos del grafo: volver a (2), elegir un nuevo vértice de partida  $v$  no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

# Grafos

## DFS (Depth First Search)

El recorrido del DFS depende del orden en que aparecen los vértices en las listas de adyacencia.



# Grafos

## DFS (Depth First Search)

```
public class Recorridos<T> {

    public void dfs(Grafo<T> grafo) {
        boolean[] marca = new boolean[grafo.listaDeVertices().tamano()+1];
        for (int i=1; i<=grafo.listaDeVertices().tamano();i++){
            if (!marca[i])
                this.dfs(i, grafo, marca);
        }
    }

    private void dfs(int i, Grafo<T> grafo, boolean[] marca){
        marca[i] = true;
        Vertice<T> v = grafo.listaDeVertices().elemento(i);
        System.out.println(v);
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);
        ady.comenzar();
        while(!ady.fin()){
            int j = ady.proximo().getVerticeDestino().getPosicion();
            if(!marca[j]){
                this.dfs(j, grafo, marca);
            }
        }
    }
}
```

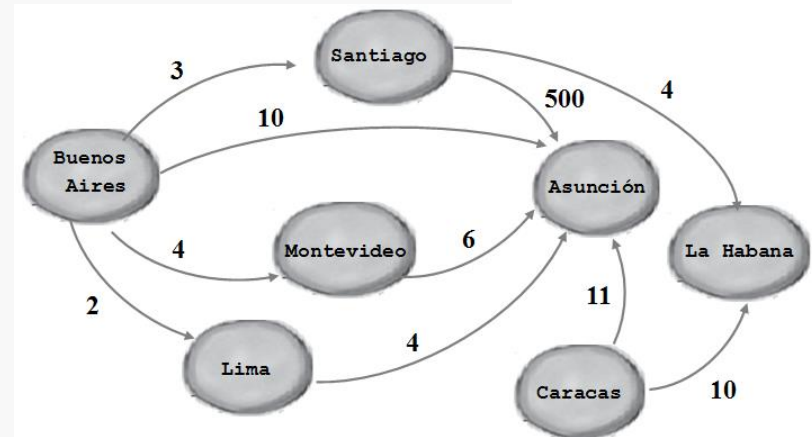
# Grafos

## DFS (Depth First Search)

### Uso del DFS

```
public class RecorridosTest {

    public static void main(String[] args) {
        Vertice<String> v1 = new VerticeImplListAdy<String>("Buenos Aires");
        Vertice<String> v2 = new VerticeImplListAdy<String>("Santiago");
        Vertice<String> v3 = new VerticeImplListAdy<String>("Lima");
        . . .
        Grafo<String> ciudades = new GrafoImplListAdy<String>();
        ciudades.agregarVertice(v1);
        ciudades.agregarVertice(v2);
        ciudades.agregarVertice(v3);
        . . .
        ciudades.conectar(v1, v2);
        ciudades.conectar(v1, v3);
        ciudades.conectar(v1, v4);
        ciudades.conectar(v1, v5);
        ciudades.conectar(v3, v5);
        . . .
        Recorridos<String> r = new Recorridos<String>();
        System.out.println("--- Se imprime el GRAFO con DFS ---");
        r.dfs(ciudades);
    }
}
```



```
Console
<terminated> RecorridosTest [Java Application] C:\Program fxe (30/05/2012)
--- Se imprime el GRAFO con DFS ---
Buenos Aires
Lima
Asuncion
Montevideo
Santiago
La Habana
Caracas
```



# Grafos

## DFS (Depth First Search)

```
public class Recorridos<T> {  
    public ListaEnlazadaGenerica<Vertice<T>> dfs(Grafo<T> grafo) {  
        boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()+1];  
        ListaEnlazadaGenerica<Vertice<T>> lis = new ListaEnlazadaGenerica<Vertice<T>>();  
        for(int i=1; i<=grafo.listaDeVertices().tamanio();i++){  
            if (!marca[i])  
                this.dfs(i, grafo, lis, marca);  
        }  
        return lis;  
    }  
    private void dfs(int i,Grafo<T> grafo,ListaEnlazadaGenerica<Vertice<T>> lis,boolean[] marca){  
        marca[i] = true;  
        Vertice<T> v = grafo.listaDeVertices().elemento(i);  
        lis.agregar(v, lis.tamanio());  
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);  
        ady.comenzar();  
        while(!ady.fin()){  
            int j = ady.proximo().getVerticeDestino().getPosicion();  
            if (!marca[j]){  
                this.dfs(j, grafo, lis, marca);  
            }  
        }  
    }  
}
```

**DFS que guarda vértices  
visitados en una lista**

# Grafos

## BFS (Breath First Search)

Este algoritmo es la generalización del recorrido por niveles de un árbol. La estrategia es la siguiente:

- Partir de algún vértice  $v$ , visitar  $v$ , después visitar cada uno de los vértices adyacentes a  $v$ .
- Repetir el proceso para cada nodo adyacente a  $v$ , siguiendo el orden en que fueron visitados.

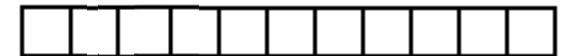
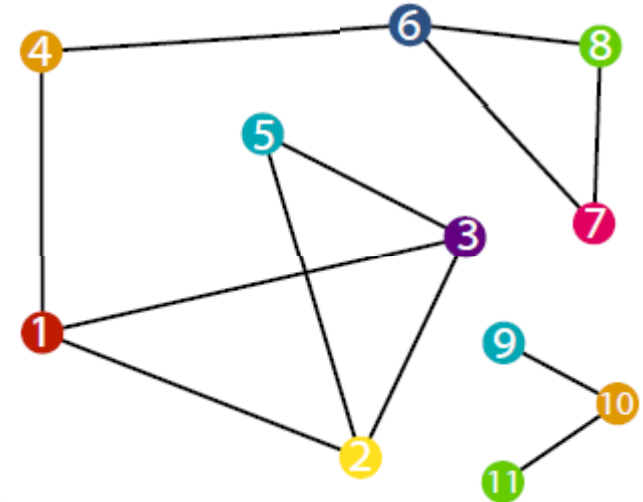
Si desde  $v$  no fueran alcanzables todos los nodos del grafo: elegir un nuevo vértice de partida no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

```
public class Recorridos {
    public void bfs(Grafo<T> grafo) {
        boolean[] marca = new boolean[grafo.listaDeVertices().tamano()+1];
        for (int i = 1; i <= marca.length; i++) {
            if (!marca[i])
                this.bfs(i, grafo, marca); //las listas empiezan en la pos 1
        }
    }
    private void bfs (int i, Grafo<T> grafo, boolean[] marca) {
        . . .
    }
}
```

# Grafos

## BFS (Breath First Search)

```
public class Recorridos {  
    ...  
    private void bfs(int i, Grafo<T> grafo, boolean[] marca) {  
        ListaGenerica<Arista<T>> ady = null;  
        ColaGenerica<Vertice<T>> q = new ColaGenerica<Vertice<T>>();  
        q.encolar(grafo.listaDeVertices().elemento(i));  
        marca[i] = true;  
        while (!q.esVacia()) {  
            Vertice<T> v = q.desencolar();  
            System.out.println(v);  
            ady = grafo.listaDeAdyacentes(v);  
            ady.comenzar();  
            while (!ady.fin()) {  
                Arista<T> arista = ady.proximo();  
                int j = arista.getVerticeDestino().getPosicion();  
                if (!marca[j]) {  
                    Vertice<T> w = arista.getVerticeDestino();  
                    marca[j] = true;  
                    q.encolar(w);  
                }  
            }  
        }  
    }  
}
```

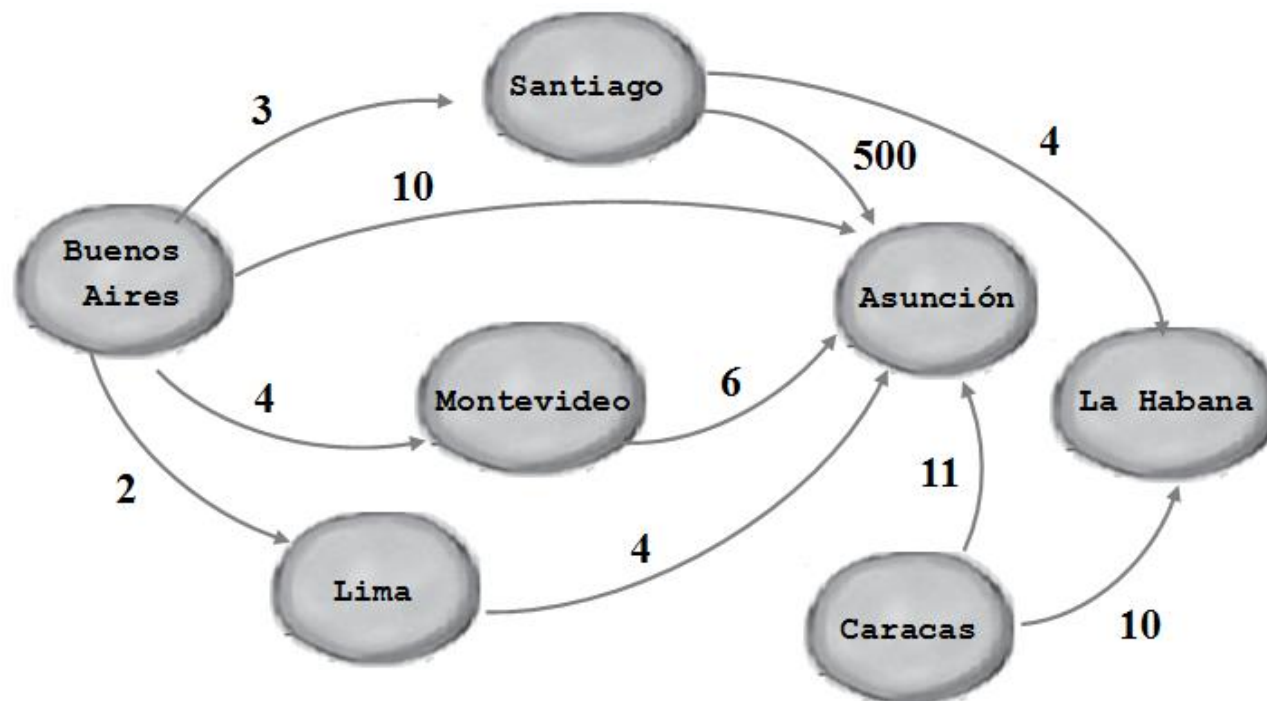


q

# Ejercicio de Parcial

Dado un Grafo orientado y valorado positivamente, como por ejemplo el que muestra la figura, implemente un método que retorne una lista con todos los caminos cuyo costo total sea igual a 10. Se considera **costo total del camino** a la suma de los costos de las aristas que forman parte del camino, desde un vértice origen a un vértice destino.

Se recomienda implementar un método público que invoque a un método recursivo privado.

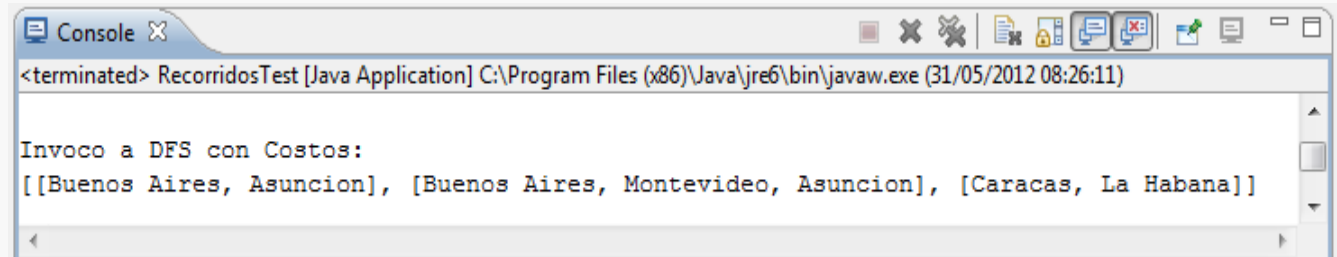
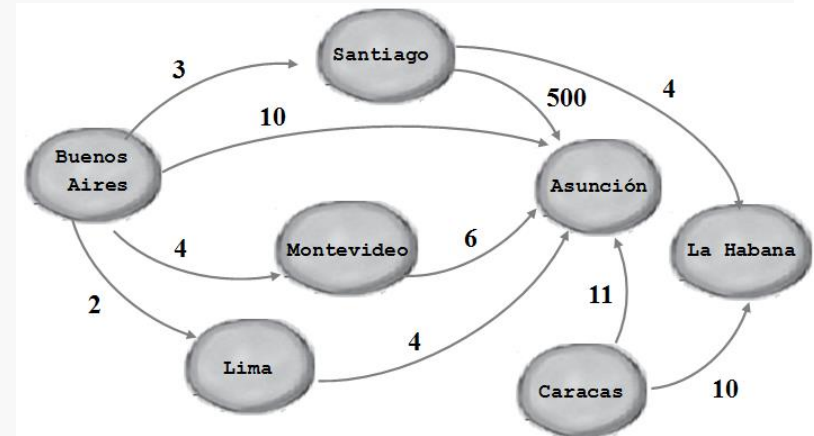


# Ejercicio de Parcial (1/2)

```
public class Recorridos {  
    public ListaGenerica<ListaGenerica<Vertice<T>>> dfsConCosto(Grafo<T> grafo) {  
        boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()+1];  
        ListaGenerica<Vertice<T>> lis = null;  
        ListaGenerica<ListaGenerica<Vertice<T>>> recorridos =  
            new ListaGenericaEnlazada<ListaGenericaEnlazada<Vertice<T>>>();  
  
        int costo = 0;  
        for(int i=1; i<=grafo.listaDeVertices().tamanio();i++){  
            lis = new ListaGenericaEnlazada<Vertice<T>>();  
            lis.add(grafo.listaDeVertices().elemento(i));  
            marca[i]=true;  
            this.dfsConCosto(i, grafo, lis, marca, costo, recorridos);  
            marca[i]=false;  
        }  
        return recorridos;  
    }  
  
    private void dfsConCosto(int i, Grafo<T> grafo, ListaGenerica<Vertice<T>> lis,  
        boolean[] marca, int costo, ListaGenerica<ListaGenerica<Vertice<T>>> recorridos) {  
        ...  
    }  
}
```

# Ejercicio de Parcial (2/2)

```
public class Recorridos {  
    private void dfsConCosto(int i, Grafo<T> grafo, ListaGenerica<Vertice<T>> lis,  
        boolean[] marca, int costo, ListaGenerica<ListaGenerica<Vertice<T>>> recorridos) {  
        Vertice<T> vDestino = null; int p=0,j=0;  
        Vertice<T> v = grafo.listaDeVertices().elemento(i);  
        ListaGenerica<Arista<T>> ady = grafo.listaDeAdyacentes(v);  
        ady.comenzar();  
        while(!ady.fin()){  
            Arista<T> arista = ady.proximo();  
            j = arista.getVerticeDestino().getPosicion();  
            if(!marca[j]){  
                p = arista.getPeso();  
                if ((costo+p) <= 10) {  
                    vDestino = arista.getVerticeDestino();  
                    lis.agregarFinal(vDestino);  
                    marca[j] = true;  
                    if ((costo+p)==10)  
                        recorridos.add(lis.copia());  
                    else  
                        this.dfsConCosto(j, grafo, lis, marca, costo+p, recorridos);  
                    lis.eliminar(vDestino);  
                    marca[j]= false;  
                }  
            }  
        }  
    }  
}
```





# Ejercicio de Parcial

## Tiempo de infección de una red

Un poderoso e inteligente virus de computadora infecta cualquier computadora en 1 minuto, logrando infectar toda la red de una empresa con cientos de computadoras. Dado un grafo que representa las conexiones entre las computadoras de la empresa, y una computadora ya infectada, escriba un programa en Java que permita determinar el tiempo que demora el virus en infectar el resto de las computadoras. Asuma que todas las computadoras pueden ser infectadas, no todas las computadoras tienen conexión directa entre sí, y un mismo virus puede infectar un grupo de computadoras al mismo tiempo sin importar la cantidad.

# Ejercicio de Parcial

```
public class BFSVirusNull {  
    public static int calcularTiempoInfeccion(Grafo<String> g, Vertice<String> inicial) {  
        int tiempo = 0;  
        boolean visitados[] = new boolean[g.listaDeVertices().tamanio()+1];  
        ColaGenerica<Vertice<String>> cola = new ColaGenerica<Vertice<String>>();  
        visitados[inicial.getPosicion()] = true;  
        cola.encolar(inicial);  
        cola.encolar(null);  
        while (!cola.esVacia()) {  
            Vertice<String> v = cola.desencolar();  
            if (v != null) {  
                ListaGenerica<Arista<String>> adyacentes = v.obtenerAdyacentes();  
                adyacentes.comenzar();  
                while (!adyacentes.fin()) {  
                    Arista<String> a = adyacentes.proximo();  
                    Vertice<String> w = a.getVerticeDestino();  
                    if (!visitados[w.getPosicion()]) {  
                        visitados[w.getPosicion()] = true;  
                        cola.encolar(w);  
                    }  
                }  
            }  
            else if (!cola.esVacia()) {  
                tiempo++;  
                cola.encolar(null);  
            }  
        }  
        return tiempo;  
    }  
}
```

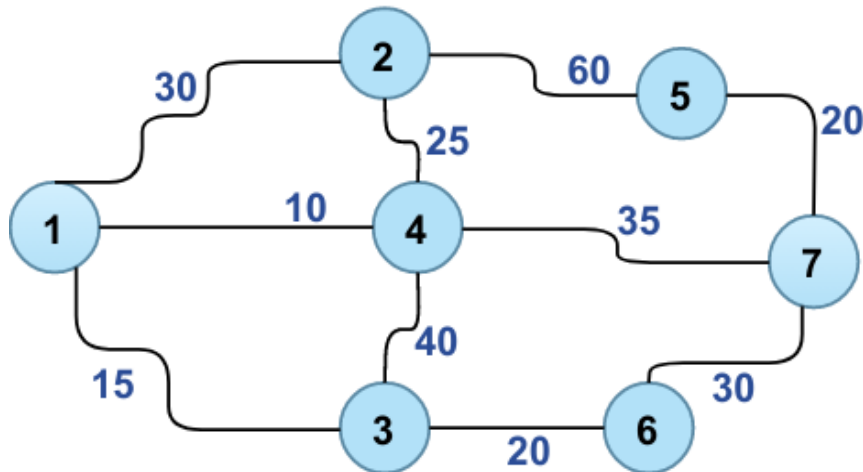
**Para pensar ...**

# Problema: El Guía de Turismo



El señor H es un guía de turismo de la ciudad de Buenos Aires. Su trabajo consiste en mostrar a grupos de turistas diferentes **puntos de interés** de la ciudad.

Estos puntos de interés están **conectados por rutas en ambos sentidos**. Dos puntos de interés vecinos tienen un servicio de bus que los conecta, con una limitación en el **número máximo de pasajeros** que puede transportar. No es siempre posible para el señor H transportar de una única vez a todos los turistas a un destino en particular.



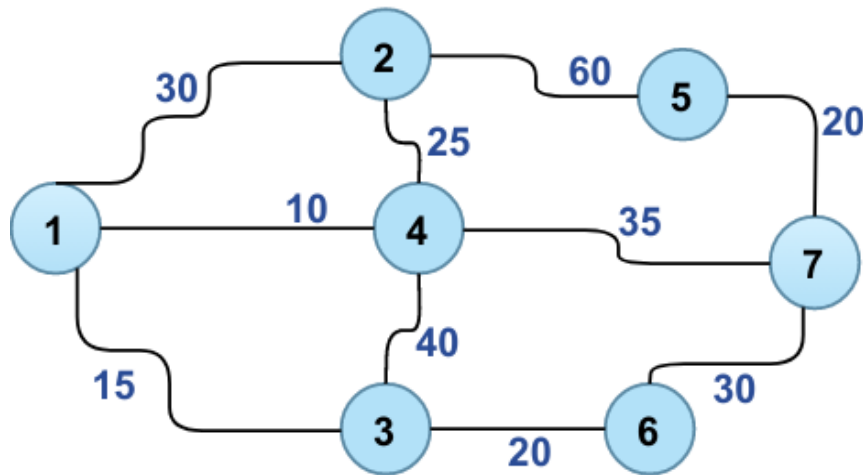
Por ejemplo, consideremos el siguiente mapa con **7 puntos de interés**, donde las aristas representan las rutas y **el peso de ellas representa el límite máximo de pasajeros a transportar por el servicio de bus**. Su misión es indicarle al Sr. H cuál es el menor número de viajes que deberá realizar para llevar al grupo de turistas de un origen a un destino.

# Problema: El Guía de Turismo



El señor H es un guía de turismo de la ciudad de Buenos Aires. Su trabajo consiste en mostrar a grupos de turistas diferentes **puntos de interés** de la ciudad.

Estos puntos de interés están **conectados por rutas en ambos sentidos**. Dos puntos de interés vecinos tienen un servicio de bus que los conecta, con una limitación en el **número máximo de pasajeros** que puede transportar. No es siempre posible para el señor H transportar de una única vez a todos los turistas a un destino en particular.

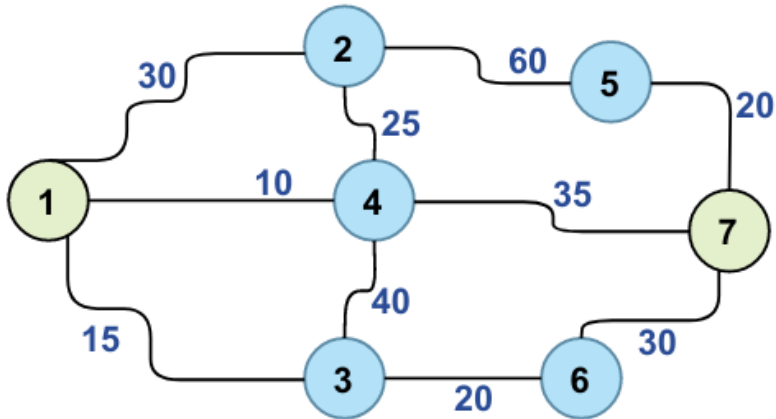


Por ejemplo, consideremos el siguiente mapa con **7 puntos de interés**, donde las aristas representan las rutas y el **peso de ellas representa el límite máximo de pasajeros a transportar por el servicio de bus**. Su misión es indicarle al Sr. H qué camino debe tomar para llevar un grupo de turistas de un origen a un destino y haciendo el menor número de viajes.

Supongamos que el señor H debe transportar a **99 turistas** del punto **1** al punto **7**.

Cuáles son los recorridos posibles?Cuál implica realizar el menor número de viajes?

# Problema: El Guía de Turismo



Puntos de interés	Cant. turistas x viaje	Cant. viajes
1 2 4 3 6 7	20	6
<b>1 2 4 7</b>	<b>25</b>	<b>5</b>
1 2 5 7	20	6
1 3 4 2 5 7	15	8
1 3 4 7	15	8
1 3 6 7	15	8
1 4 2 5 7	10	11
1 4 3 6 7	10	11
1 4 7	10	11



# Problema: Entregar sueldo a jubilados

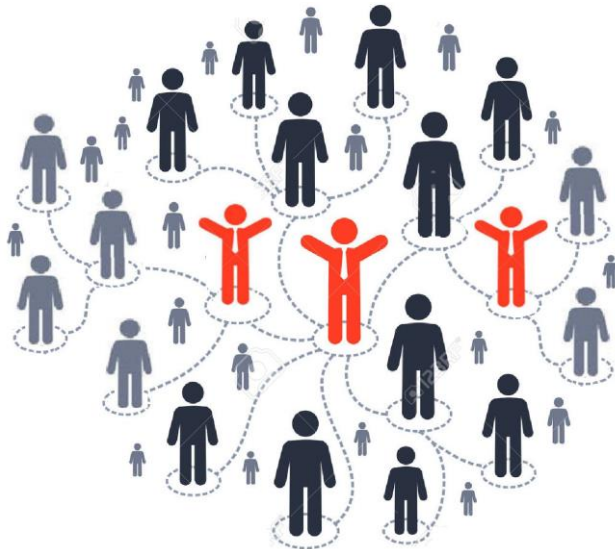


El **Banco Itaú** se suma a las campañas "**QUEDATE EN CASA**" lanzando un programa para acercarle a los jubilados el sueldo hasta sus domicilios. Para ello el banco cuenta con información que permite definir un grafo de personas donde la persona puede ser un jubilado o un empleado del banco que llevará el dinero.

Se necesita armar la cartera de jubilados para que cada empleado repartidor del banco, incluyendo en cada lista los jubilados que vivan un radio cercano a su casa y no hayan percibido la jubilación del mes.

Para ello, implemente un algoritmo que dado un **Grafo<Persona>** retorne una lista de jubilados que se encuentren a una distancia menor a un valor dado del empleado Itaú (grado de separación del empleado Itaú).

El método recibirá un **Grafo<Persona>**, un **empleado** y un **grado de separación/distancia** y debe retornar una lista de hasta 40 jubilados que no hayan percibido la jubilación del mes y se encuentren a una distancia menor a recibido como parámetro.



En este grafo simple, donde los empleados del banco están en color rojo y se desea retornar los jubilados hasta distancia 2, se debería retornar los los jubilados color negro.

La **Persona** conoce si es empleado o jubilado, el nombre, domicilio.