UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos

Práctica 5
Grafos

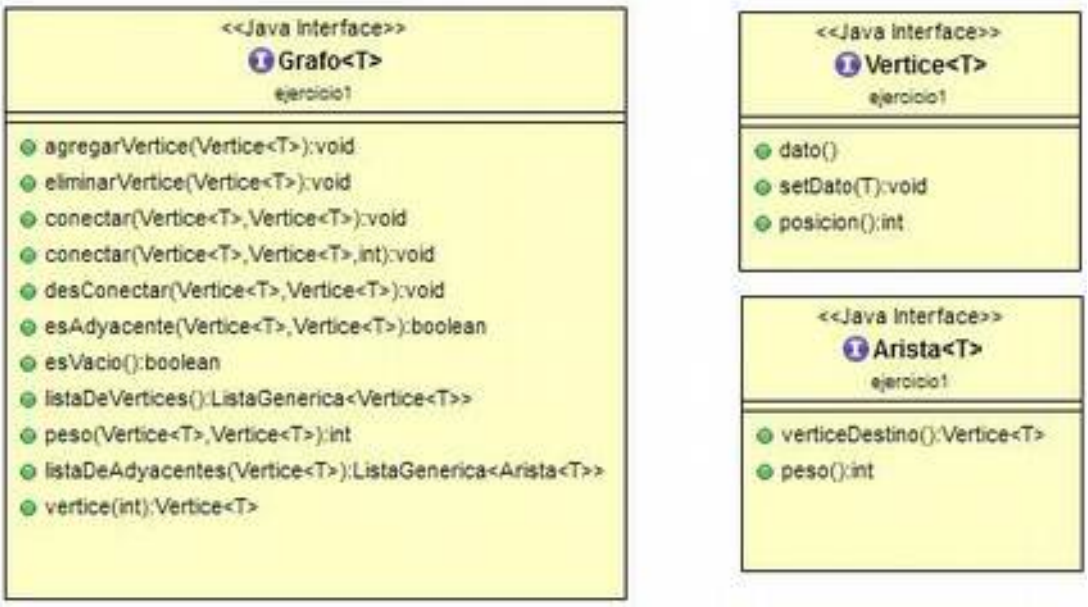
Ejercicio 1
Teniendo en cuenta las dos representaciones de grafos: Matriz de Adyacencias y Lista de Adyacencias.

a. Bajo qué condiciones usaría una Matriz de Adyacencias en lugar de una Lista de Adyacencias para representar un grafo. Y una Lista de Adyacencias en lugar de una Matriz de Adyacencias. **Fundamentar**.
b. ¿En función de qué parámetros resulta apropiado realizar la estimación del orden de ejecución para algoritmos sobre grafos densos? ¿Y para algoritmos sobre grafos dispersos? **Fundamentar**.
c. Si representamos un grafo no dirigido usando una Matriz de Adyacencias, ¿cómo sería la matriz resultante? **Fundamentar**.

Ejercicio 2

a. Responda las siguientes preguntas considerando un grafo no dirigido de n vértices. **Fundamentar**.
    i. ¿Cuál es el mínimo número de aristas que puede tener si se exige que el grafo sea conexo?
    ii. ¿Cuál es el máximo número de aristas que puede tener si se exige que el grafo sea acíclico?
    iii. ¿Cuál es el número de aristas que puede tener si se exige que el grafo sea conexo y acíclico?
    iv. ¿Cuál es el número de aristas que puede tener si se exige que el grafo sea completo? (Un grafo es completo si hay una arista entre cada par de vértices.)
b. En un grafo dirigido y que no tiene aristas que vayan de un nodo a sí mismo, ¿Cuál es el mayor número de aristas que puede tener? **Fundamentar**.

Ejercicio 3
Sea la siguiente **especificación** de un Grafo:



Interface Grafo

- El método **agregarVertice(Vertice<T> v)** //Agrega un vértice al Grafo. Verifica que el vértice no exista en el Grafo.
- El método **eliminarVertice(Vertice<T> v)** // Elimina el vértice del Grafo. En caso que el vértice tenga conexiones con otros vértices, se eliminan todas sus conexiones.
- El método **conectar(Vertice<T> origen, Vertice<T> destino)** //Conecta el vértice *origen* con el vértice *destino*. Verifica que ambos vértices existan, caso contrario no realiza ninguna conexión.
- El método **conectar(Vertice<T> origen, Vertice<T> destino, int peso)** // Conecta el vértice *origen* con el vértice *destino* con *peso*. Verifica que ambos vértices existan, caso contrario no realiza ninguna conexión.
- El método **desConectar(Vertice<T> origen, Vertice<T> destino)** //Desconecta el vértice *origen* con el *destino*. Verifica que ambos vértices y la conexión *origen -->destino* existan, caso contrario no realiza ninguna desconexión. En caso de existir la conexión *destino -->origen*, ésta permanece sin cambios.
- El método **esAdyacente(Vertice<T> origen, Vertice<T> destino): boolean** // Retorna true si *origen* es adyacente a *destino*. False en caso contrario.
- El método **esVacio(): boolean** // Retorna true en caso que el grafo no contenga ningún vértice. False en caso contrario.
- El método **listaDeVertices(): ListaGenerica<Vertice<T>>** //Retorna la lista con todos los vértices del grafo.
- El método **peso(Vertice<T> origen, Vertice<T> destino): int //**Retorna el peso de la conexión *origen -->destino* . Si no existiera la conexión retorna 0.
- El método **listaDeAdyacentes(Vertice<T> v)): ListaGenerica<Arista**> // Retorna la lista de adyacentes de un vértice.
- El método **vertice(int posicion): Vertice<T>** // Retorna el vértice dada su posición.

Interface Vértice

- El método **dato(): T** // Retorna el dato del vértice.
- El método **setDato(T d)** // Setea el dato del vértice

- El método **posicion(): int** // Retorna la posición del vértice.

Interface Arista
- El método **verticeDestino(): Vertice<T>** // Retorna el vértice destino de la arista.
- El método **peso(): int** // Retorna el peso de la arista

a) Defina las interfaces **Grafo**, **Vértice** y **Arista** de acuerdo a la especificación que se detalló, ubicada dentro del paquete ejercicio1.

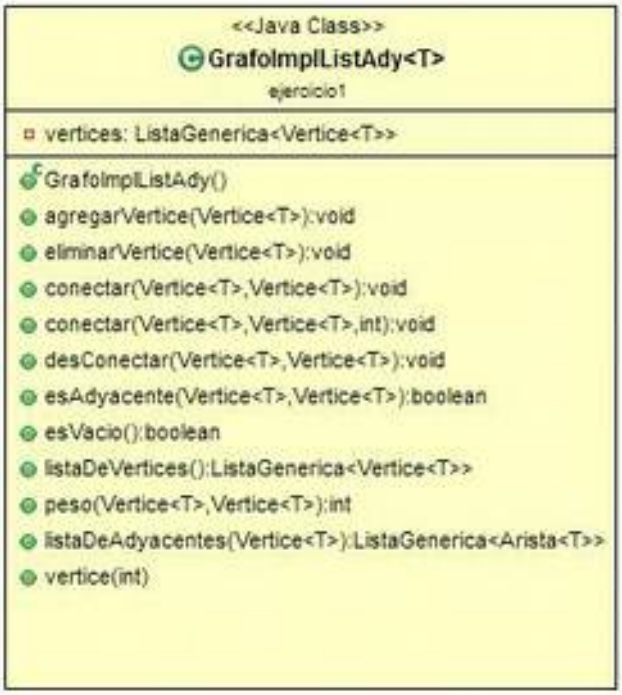
b) Escriba una clase llamada **GrafoImplMatrizAdy** que implemente la interface **Grafo**, y una clase llamada **VerticeImplMatrizAdy** que implemente la interface **Vértice**.

c) Escriba una clase llamada **GrafoImplListAdy** que implemente la interface **Grafo**, y una clase llamada **VerticeImplListAdy** que implemente la interface **Vértice**.

d) Escriba una clase llamada **AristaImpl** que implemente la interface **Arista.** Es posible utilizar la interface y clases que implementan la misma tanto para grafos ponderados como no ponderados? Analice el comportamiento de los métodos que componen la misma.

e) Analice qué métodos cambiarían el comportamiento en el caso de utilizarse para modelar grafos dirigidos.

Nota: la clase **ListaGenerica** es la utilizada previamente.


<<Java Class>>
GrafoImplListAdy<T>
ejercicio1
vertices: ListaGenerica<Vertice<T>>
GrafoImplListAdy()
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
esVacio():boolean
listaDeVertices():ListaGenerica<Vertice<T>>
peso(Vertice<T>,Vertice<T>):int
listaDeAdyacentes(Vertice<T>):ListaGenerica<Arista<T>>
vertice(int)


<<Java Class>>
VerticeImplListAdy<T>
ejercicio1
adyacentes: ListaGenerica<Arista<T>>
dato: T
posicion: int
VerticeImplListAdy(T)
dato()
setDato(T):void
posicion():int
conectar(Vertice<T>):void
conectar(Vertice<T>,int):void
desconectar(Vertice<T>):void
obtenerAdyacentes():ListaGenerica<Arista<T>>
esAdyacente(Vertice<T>):boolean
peso(Vertice<T>):int
obtenerArista(Vertice<T>):Arista<T>
setPosicion(int):void


<<Java Class>>
AristaImpl<T>
ejercicio1
destino: Vertice<T>
peso: int
AristaImpl(Vertice<T>,int)
verticeDestino():Vertice<T>
peso():int

Ejercicio 4

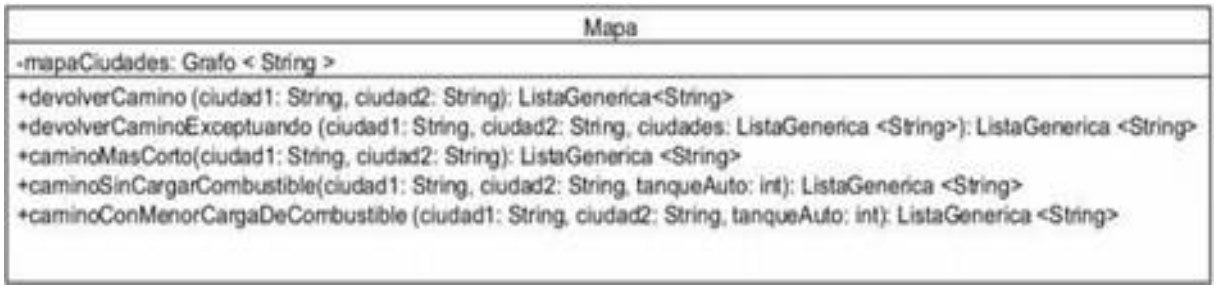
a. Implemente en JAVA una clase llamada **Recorridos** ubicada dentro del paquete **ejercicio5** cumpliendo la siguiente especificación:

dfs(Grafo<T> grafo): ListaGenerica <T> // Retorna una lista de vértices con el recorrido en profundidad del *grafo* recibido como parámetro.

bfs(Grafo<T> grafo): ListaGenerica <T>// Retorna una lista de vértices con el recorrido en amplitud del *grafo* recibido como parámetro.

b. Estimar los órdenes de ejecución de los métodos anteriores.

Ejercicio 5

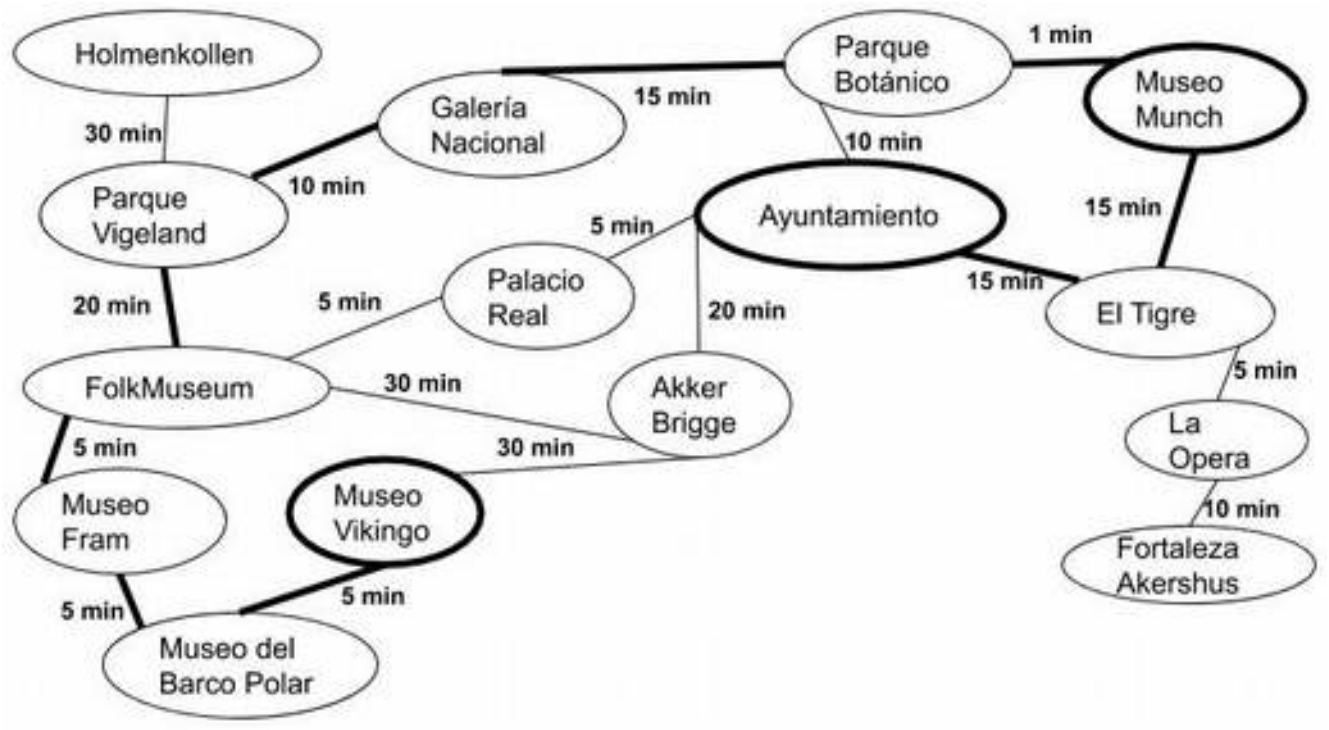


- El método **devolverCamino (String ciudad1, String ciudad2): ListaGenerica<String>** // Retorna la lista de ciudades que se deben atravesar para ir de *ciudad1* a *ciudad2* en caso que se pueda llegar, si no retorna la lista vacía. (Sin tener en cuenta el combustible).
- El método **devolverCaminoExceptuando (String ciudad1, String ciudad2, ListaGenerica<String> ciudades): ListaGenerica<String>** // Retorna la lista de ciudades que forman un camino desde *ciudad1* a *ciudad2*, sin pasar por las ciudades que están contenidas en la lista *ciudades* pasada por parámetro, si no existe camino retorna la lista vacía. (Sin tener en cuenta el combustible).
- El método **caminoMasCorto(String ciudad1, String ciudad2): ListaGenerica<String>** // Retorna la lista de ciudades que forman el camino más corto para llegar de *ciudad1* a *ciudad2*, si no existe camino retorna la lista vacía. (Las rutas poseen la distancia). (Sin tener en cuenta el combustible).
- El método **caminoSinCargarCombustible(String ciudad1, String ciudad2, int tanqueAuto): ListaGenerica<String>** // Retorna la lista de ciudades que forman un camino para llegar de *ciudad1* a *ciudad2*. El auto no debe quedarse sin combustible y no puede cargar. Si no existe camino retorna la lista vacía.
- El método **caminoConMenorCargaDeCombustible (String ciudad1, String ciudad2, int tanqueAuto): ListaGenerica<String>** // Retorna la lista de ciudades que forman un camino para llegar de *ciudad1* a *ciudad2* teniendo en cuenta que el auto debe cargar la menor cantidad de veces. El auto no se debe quedar sin combustible en medio de una ruta, además puede completar su tanque al llegar a cualquier ciudad. Si no existe camino retorna la lista vacía.

UNLP. Facultad de Informática.
Algoritmos y Estructuras de Datos

Ejercicio 6
Se quiere realizar un paseo en bicicleta por lugares emblemáticos de Oslo. Para esto se cuenta con un grafo de bicisendas. Partiendo desde el "Ayuntamiento" hasta un lugar destino en menos de X minutos, sin pasar por un conjunto de lugares que están restringidos.
Escriba una clase llamada **VisitaOslo** e implemente su método:

ListaGenerica<String> paseoEnBici(Grafo<String> lugares, String destino, int maxTiempo, ListaGenerica<String> lugaresRestringidos)



En este ejemplo, para llegar desde **Ayuntamiento** a **Museo Vikingo**, sin pasar por: {"Akker Brigge", "Palacio Real"} y en no más de 120 minutos, el camino marcado en negrita cumple las condiciones.
Notas:

- El "Ayuntamiento" debe ser buscado antes de comenzar el recorrido para encontrar un camino.
- De no existir camino posible, se debe retornar una lista vacía.
- Debe retornar el **primer camino** que encuentre que cumple las restricciones.
- Ejemplos de posibles caminos a retornar, sin pasar por "Akker Brigge" y "Palacio Real" en no más de 120 min (maxTiempo)
    - **Ayuntamiento, El Tigre, Museo Munch, Parque Botánico, Galería Nacional, Parque Vigeland, FolkMuseum, Museo Fram, Museo del Barco Polar, Museo Vikingo. El recorrido se hace en 91 minutos.**

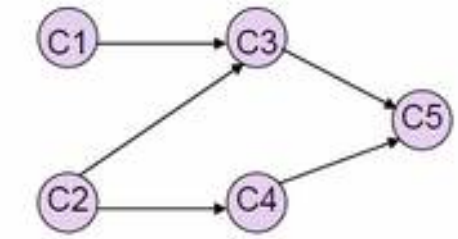
- **Ayuntamiento, Parque Botánico, Galería Nacional, Parque Vigeland, FolkMuseum, Museo Fram, Museo del Barco Polar, Museo Vikingo. El recorrido se hace en 70 minutos.**

Ejercicio 7 - Ordenación Topológica

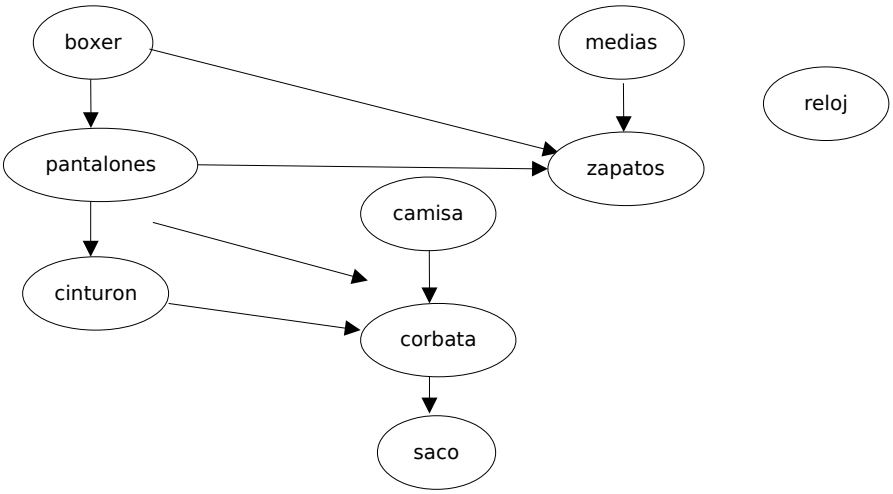
La organización topológica (o "sort topológico") de un grafo dirigido acíclico (DAG) es un proceso de asignación de un orden lineal a los vértices del DAG de modo que si existe una arista (v,w) en el DAG, entonces v aparece antes de w en dicho ordenamiento lineal.

Por ejemplo, sea el siguiente DAG, posibles organizaciones topológicas son las siguientes:

- C1, C2, C4, C3 y C5
- C2, C4, C1, C3 y C5
- C1, C2, C3, C4 y C5



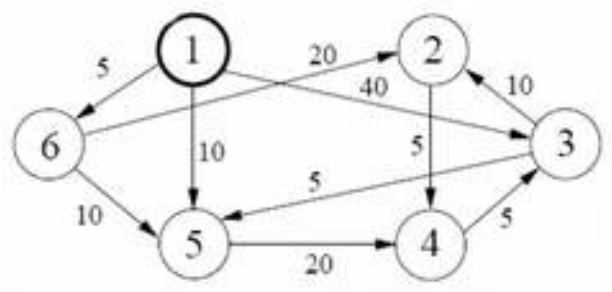
El siguiente DAG surge cuando el Profesor Miguel se viste a la mañana. El profesor debe ponerse ciertas prendas antes que otras. Por ejemplo, las medias antes que los zapatos. Otras prendas pueden ponerse en cualquier orden. Por ejemplo, las medias y los pantalones. Una arista dirigida (v,w) en el DAG indica que la prenda v debe ser puesta antes que la prenda w. Enumere algunos posibles órdenes topológicos que se pueden obtener a partir del DAG previo.



Ejercicio 8 – Algoritmo de Dijkstra

El algoritmo de Dijkstra que permite encontrar el camino mínimo desde un cierto nodo hacia todos los demás. Para ello, el algoritmo verifica con cada uno de los nodos si utilizando ese nodo como nodo intermedio, puede mejorar la distancia al resto de los nodos.

Sea el siguiente Dígrafo, considerando al nodo 1 como origen, la tabla inicial es la siguiente.

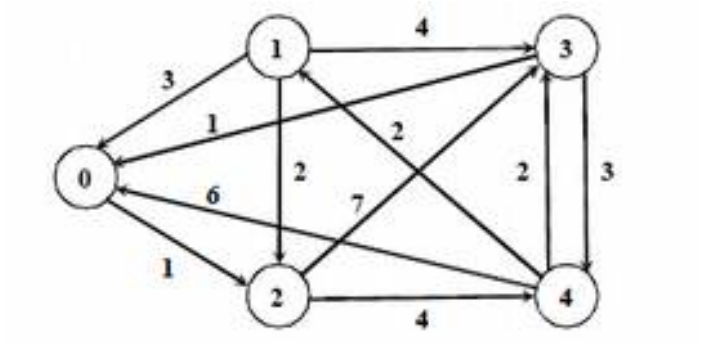


V	D_v	P_v	Conoc.
1	0	0	(1)
2	∞	0	0
3	**40**	**1**	0
4	∞	0	0
5	**10**	**1**	0
6	**5**	**1**	0

La secuencia de nodos a elegir es la siguiente: 6, 5, 2, 4 y 3. De esta forma, se obtiene la siguiente tabla final.

V	D_v	P_v	Conoc.
1	0	0	1
2	25	6	1
3	35	4	1
4	30	5	1
5	10	1	1
6	5	1	1

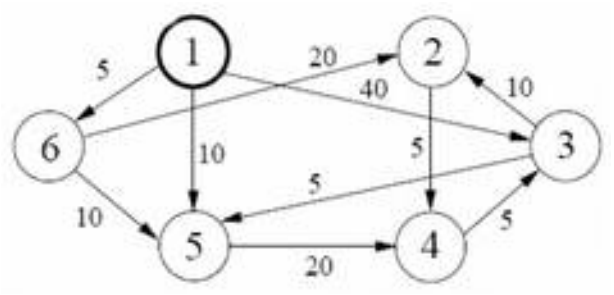
Sea el siguiente dígrafo:



a) Para el vértice inicial 3, describa paso a paso la ejecución del algoritmo, mostrando como varían los costos de acceso desde el vértice inicial a cada uno de los vértices restantes.
b) Muestre mediante un ejemplo como falla el algoritmo de Dijkstra si existen en el dígrafo aristas de costo negativo.
c) El algoritmo de Dijkstra se puede implementar de 2 formas distintas en función de cómo se identifica al vértice que se utiliza como pivote para verificar si pasando por ese vértice se puede reducir el costo de llegar a cada uno de los demás. Describa las dos formas (no tiene que implementarlas) e indique el tiempo de ejecución de cada una.

Ejercicio 9 - Algoritmo de Floyd

El algoritmo de Floyd permite encontrar el camino mínimo para cada par de vértices. Esto lo realiza verificando sistemáticamente si para cada par de vértices puede reducir el costo pasando por cada uno de los restantes vértices. Sea el siguiente dígrafo, la tabla inicial de costos es la siguiente.

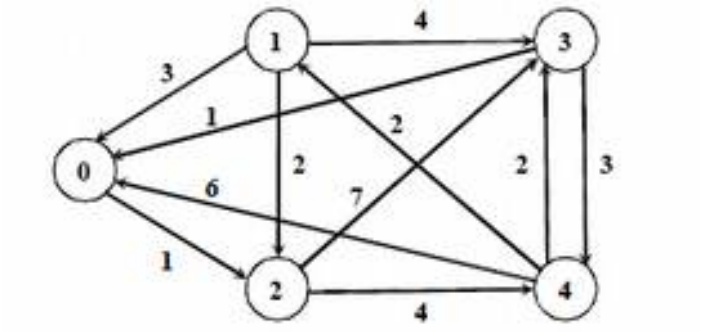


	1	2	3	4	5	6
1	-	∞	40	∞	10	5
2	∞	-	∞	5	∞	∞
3	∞	10	-	∞	5	∞
4	∞	∞	5	-	∞	∞
5	∞	∞	∞	20	-	∞
6	∞	20	∞	∞	10	-

El resultado del algoritmo de Floyd es la siguiente tabla de costos.

	1	2	3	4	5	6
1	-	25	35	30	10	5
2	∞	-	10	5	15	∞
3	∞	10	-	15	5	∞
4	∞	15	5	-	10	∞
5	∞	35	25	20	-	∞
6	∞	20	30	25	10	-

Sea el siguiente dígrafo, describa paso a paso la ejecución del algoritmo.



Ejercicio 10

Se desea mantener un conjunto de antenas situadas estratégicamente por una zona determinada. Se conoce cuál es el costo de ir de una antena a otras antenas

cercanas. El equipo de mantenimiento trata de optimizar las rutas de visita a las antenas de forma que el costo de mantener las antenas sea mínimo.

El mapa de antenas junto con el costo de ir de unas antenas a otras lo representaremos en la siguiente matriz:

	Antena 1	Antena 2	Antena 3	Antena 4	Antena 5	Antena 6	Antena 7
Antena 1	0	7	2	6	9		8
Antena 2	7	0		3			
Antena 3	2		0		6		
Antena 4	6	3		0			3
Antena 5	9		6		0	3	
Antena 6					3	0	2
Antena 7	8			3		2	0

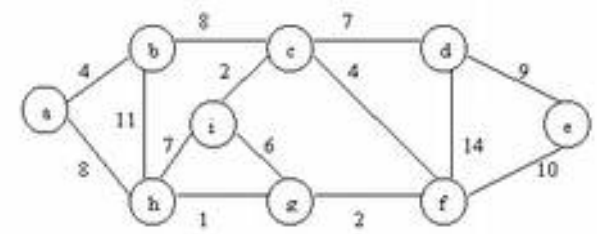
Cuando no aparece valor entre dos antenas es porque no se puede llegar directamente desde una a la otra.

a) ¿Qué algoritmo se puede aplicar para calcular el costo mínimo para ir desde la antena 1 hasta la antena 7?
b) Muestre el árbol de caminos mínimos desde la antena 1 hacia todas las demás.

Ejercicio 11 - Algoritmo de Prim

El algoritmo de Prim consiste en ir construyendo un árbol haciéndolo crecer por etapas, considerando en cada etapa la arista de menor costo de forma tal de que solo uno de los nodos de esa arista pertenezca al árbol.

La implementación del algoritmo de Prim es similar a la del algoritmo de Dijkstra, registrando la información en una tabla. Muestre paso a paso la aplicación del algoritmo de Prim al siguiente Grafo.



Ejercicio 12 - Algoritmo de Kruskal

El algoritmo de Kruskal consiste en ir seleccionando aristas del Grafo si la misma no produce un ciclo. A diferencia de Prim, en Kruskal no se debe ir armando un árbol, sino que se puede considerar cualquier arista mientras que no origine un ciclo. Muestre paso a paso la aplicación del algoritmo de Kruskal al siguiente Grafo.