Programación III
TEMA 8: Grafos
Práctica nº 8 - A

Ejercicio 1

a. ¿Cuál es la diferencia entre un grafo y un árbol?
b. Indicar para cada uno de los siguientes casos si la relación se representa a través de un **grafo no dirigido** o de un **grafo dirigido** (digrafo).
    i. Vértices: países. Aristas: es limítrofe.
    ii. Vértices: países. Aristas: principales mercados de exportación.
    iii. Vértices: dispositivos en una red de computadoras. Aristas: conectividad.
    iv. Vértices: variables de un programa. Aristas: relación “usa”. (Decimos que la variable **x** usa la variable **y**, si **y** aparece del lado derecho de una expresión y **x** aparece del lado izquierdo, por ejemplo **x = y**).

Ejercicio 2

a. En un grafo no dirigido de n vértices, ¿Cuál es el número de aristas que puede tener? **Fundamentar**.
    i. ¿Cuál es el mínimo número de aristas que puede tener 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 mínimo 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**.

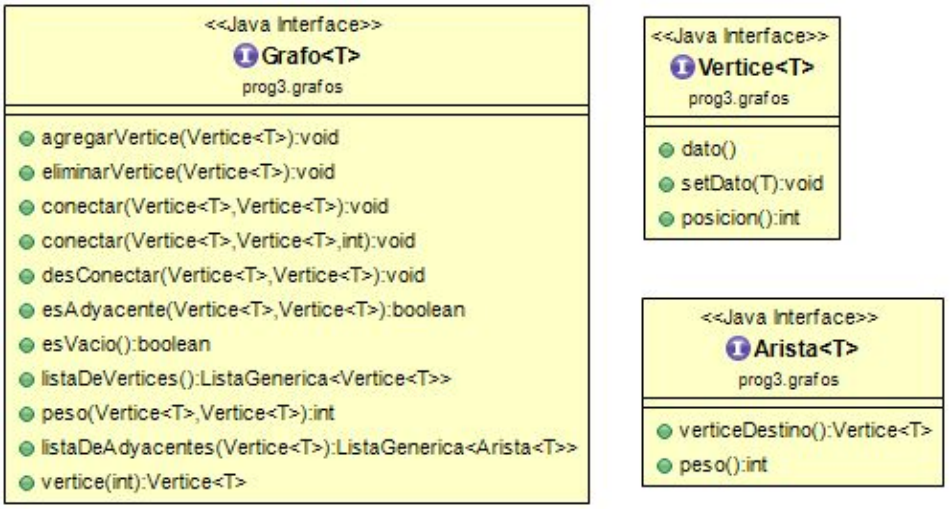
Ejercicio 3

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 4

A continuación la **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.

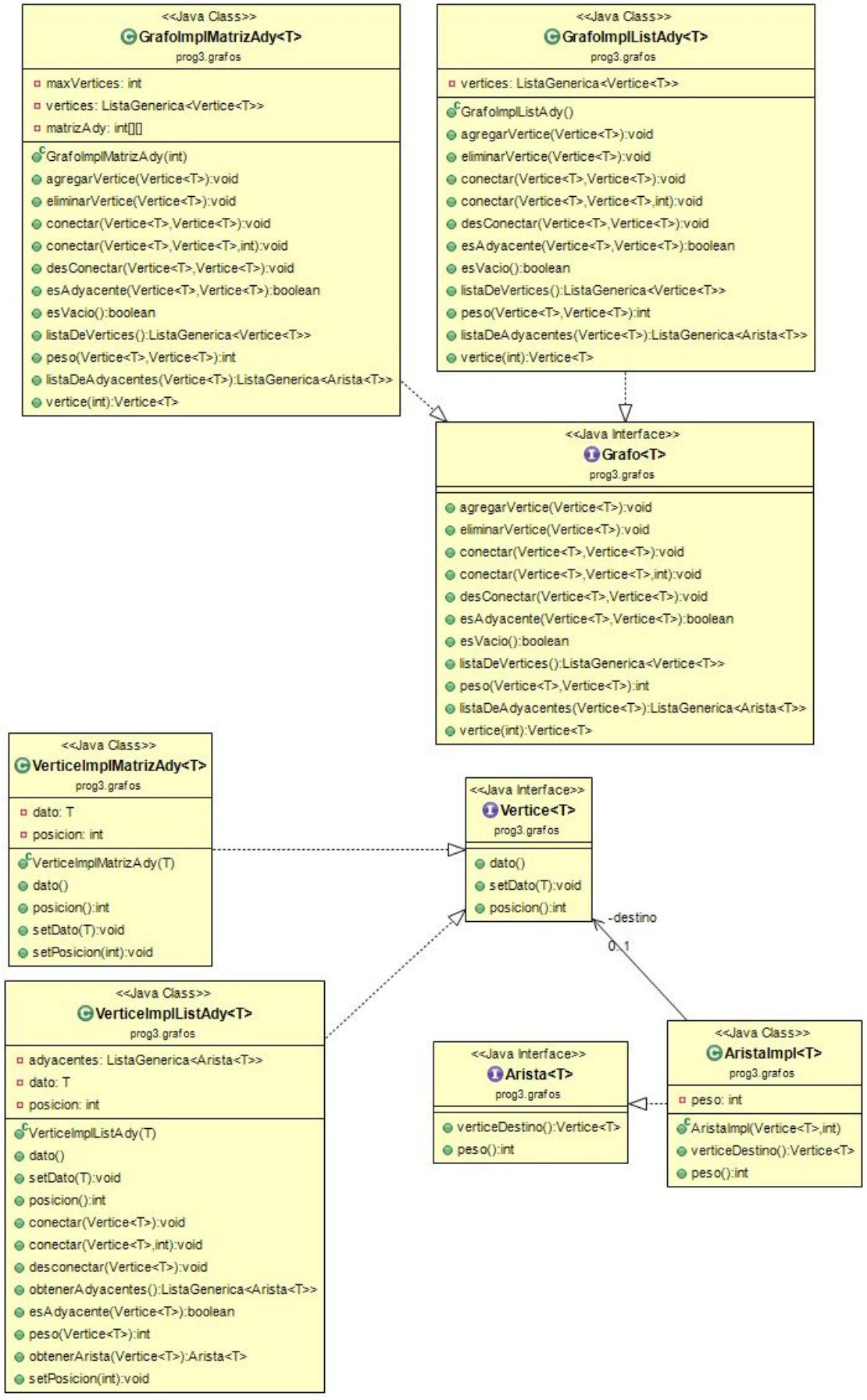
Interface Vértice

- 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 continuación se muestra el diagrama de clases de la implementación de estas interfaces:



Copie las clases provistas por la cátedra (en el .zip están contenidas en el paquete prog3.grafos y de ser necesario la implementación de lista genérica). Además:

a. **Analice qué métodos cambiarían el comportamiento** en el caso de utilizarse para modelar grafos NO dirigidos.
b. **Verifique las implementaciones** de las clases GrafoImplMatrizAdy y GrafoImplListAdy con las clases de Test JUnits provistas por la cátedra.

Ejercicio 5

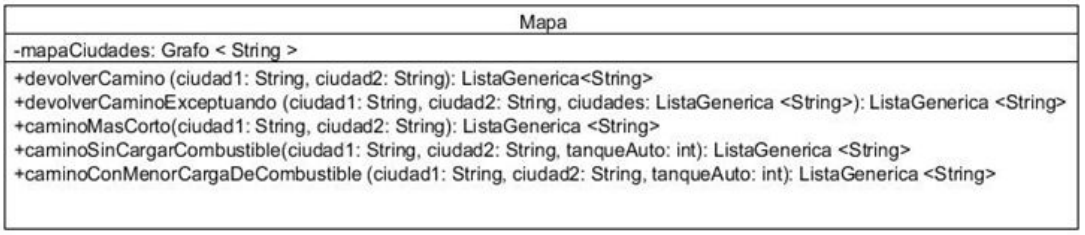
a. Implemente en JAVA una clase llamada **Recorridos** ubicada dentro del paquete **prog3.grafos.utiles**, cumpliendo la siguiente especificación:

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

bfs(Grafo<T> grafo): ListaGenerica<Vertice<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 6



- 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.

Ejercicio 7

Implemente en JAVA una clase llamada **Algoritmos** ubicada dentro del paquete **prog3.grafos.utiles**, cumpliendo la siguiente especificación:

- **subgrafoCuadrado(Grafo<T> grafo): boolean** // Retorna *true* si un dígrafo contiene un subgrafo cuadrado, *false* en caso contrario. Un subgrafo cuadrado es un ciclo simple de longitud 4.
- **getGrado(Grafo<T> grafo) : int** // Retorna el **grado** del digrafo pasado como parámetro. El grado de un digrafo es el de su vértice de grado máximo. El grado de un vértice en un grafo dirigido es la suma del número de aristas que salen de él (grado de salida) y el número de aristas que entran en él (grado de entrada).
- **tieneCiclo(Grafo<T> grafo): boolean** // Retorna *true* si el grafo dirigido pasado como parámetro tiene al menos un ciclo, *false* en caso contrario.

Ejercicio 8

El delta está compuesto por una gran cantidad de canales de agua que unen pequeñas islas. Cada isla está comunicada con otras por medio de uno o más canales estrechos de agua, pero lo suficientemente amplios como para que pasen dos lanchas (una en cada sentido). Las lanchas inter-isleñas tienen una tarifa única y salen todas del muelle principal. Cuando un pasajero se embarca, puede descender en cualquier isla, recorrer los atractivos de la misma y luego volver al muelle de la isla a tomar la próxima lancha para seguir su recorrido utilizando el mismo boleto. Dicho boleto tiene validez hasta que el pasajero vuelve al muelle principal, momento en el cual termina su recorrido.

Si un pasajero quiere salir en otra lancha por otro brazo de islas, deberá comprar otro boleto.

Cada tramo tiene un cartel indicador de la distancia en metros que existe entre las dos islas que comunica.

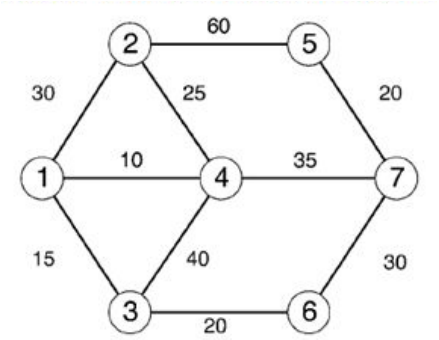
Implemente en JAVA una clase llamada **Delta** ubicada dentro del paquete **prog3.grafos.utiles**, cumpliendo la siguiente especificación:

- **maxIslasDistintas(Grafo<String> grafo) : int** // Retorna el número máximo de islas distintas que se pueden recorrer en el *grafo* comprando un único boleto.
- **caminoMasCorto(Grafo<String> grafo, String islaO, String islaD) : RutaMinima** // Retorna un objeto de la clase RutaMinima, el cual contiene el camino más corto entre *islaO* e *islaD* y si se puede realizar con un único boleto o es necesario comprar un nuevo boleto para completar el recorrido.

Ejercicio 9 – 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 número sobre ellas representa el límite máximo de pasajeros a transportar por el servicio de bus.



En este ejemplo, para transportar a 99 turistas de La ciudad 1 a La ciudad 7, le tomará 5 viajes, eligiendo la ruta con menor número de viajes a realizar: 1 - 2 - 4 - 7. (En cada viaje el servicio de bus puede transportar como máximo a 25 pasajeros, 24 turistas + al señor H, en los cuatro primeros viajes transporta a 96 turistas y en el último a los restantes 3).

Implemente en JAVA una clase llamada **GuiaDeTurismo** ubicada dentro del paquete **prog3.grafos.utiles**, cumpliendo la siguiente especificación:

- **caminoConMenorNrodeViajes(Grafo<String> grafo, String puntoInteresOrigen, String puntoInteresDestino) : ListaGenerica <String>** // Retorna la lista de puntos de interés que se deben atravesar en el *grafo* para ir de *puntoInteresOrigen* a *puntoInteresDestino*, haciendo la menor cantidad de viajes.

Ejercicio 10 – Grados de Separación

En nuestro interconectado mundo se especula que dos personas cualesquiera están relacionadas entre sí a lo sumo por 6 grados de separación. En este problema, debemos realizar un método para encontrar el **máximo grado de separación** en una red de personas, donde una arista entre dos personas representa la relación de conocimiento entre ellas, la cual es simétrica.

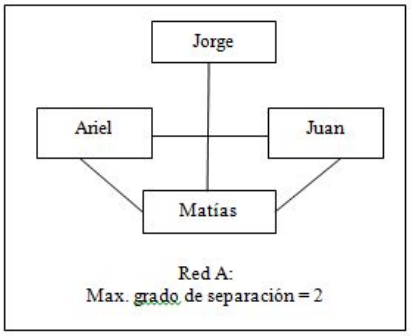
Entre dos personas, el grado de separación es el mínimo número de relaciones que son necesarias para conectarse entre ellas.

Si en la red hay dos personas que no están conectadas por una cadena de relaciones, el grado de separación entre ellas se considerará igual a -1.

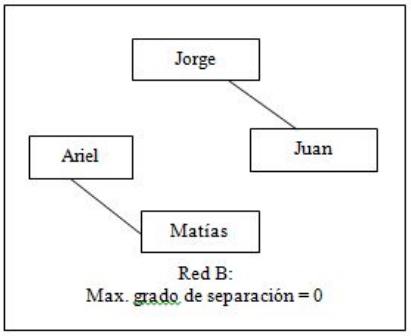
En una red, el **máximo grado de separación** es el mayor grado de separación entre dos personas cualesquiera de la red.

Implemente en JAVA una clase llamada **GradosDeSeparacion** ubicada dentro del paquete **prog3.grafos.utiles**, cumpliendo la siguiente especificación:

- **maximoGradoDeSeparacion(Grafo<String> grafo) : int** // Retorna el máximo grado de separación del grafo recibido como parámetro. Si en el grafo hubiera dos personas cualesquiera que no están conectadas por una cadena de relaciones entonces se retorna 0.



Red A:
Max. grado de separación = 2



Red B:
Max. grado de separación = 0