

## Examen Recuperatorio

### 2do Parcial

Tiempo mínimo para el examen: 1 hora reloj.

Tiempo máximo para el examen: 2 horas reloj.

1. Considere la siguiente clase `_Nodo`:

```
class _Nodo:
    def __init__(self, dato: Any = None, prioridad: int = 0,
                 siguiente: '_Nodo' = None):
        self.dato = dato
        self.prioridad = prioridad
        self.sig = siguiente
```

Utilícela para completar la implementación el método `agregar_elemento` de la siguiente clase:

```
class ColaPrioridad:
    def __init__(self):
        self.cabeza = None
        self.longitud = 0

    def agregar_elemento(self, dato: Any, prioridad: int) -> None:
        '''Este método agrega el dato junto con su prioridad en la
        cola de prioridad de manera ordenada, dejando los datos de mayor
        prioridad primeros en la cola. Se considera el valor 0
        como la mayor prioridad, seguida de una prioridad 1,
        luego la 2, etc. Si se quiere ingresar un dato con una
        prioridad ya existente en la cola, se respeta el orden de
        llegada.'''
        pass

    def quitar_elemento(self) -> Any:
        if self.cabeza is None:
            return None
        dato = self.cabeza.dato
        self.cabeza = self.cabeza.sig
        self.longitud -= 1
        return dato
```

2. a. Considere la clase `GrafoDirigido1` que sirve para representar un grafo dirigido:

```
class GrafoDirigido1:
    def __init__(self):
        self.vertices: list[Any] = [] #lista de vertices grafo dirigido
        self.aristas: list[(Any, Any)] = [] #lista que contiene todas las
                                           #aristas dirigidas del grafo

    def grado_salida(self, v: Any) -> int:
        ''' Recibe un vertice y devuelve el número de aristas
        que salen de él'''
        pass
```

```

def grado_entrada(self, v: Any) -> int:
    ''' Recibe un vertice y devuelve el número de aristas
    que llegan a él'''
    pass

def vertices_siguientes(self, v: Any) -> list[Any]:
    ''' Recibe un vertice y devuelve una lista de todos los vertices
    a los cuales se puede llegar a través de una arista saliente'''
    pass

```

Implemente los métodos `grado_salida`, `grado_entrada` y `vertices_siguientes` de esta clase `GrafoDirigido1`.

- b. Considere ahora esta otra clase `GrafoDirigido2` que sirve para representar un grafo dirigido:

```

class GrafoDirigido2:
    def __init__(self):
        self.vertices: list[Any] = [] #lista de vertices grafo dirigido
        self.vecinos: dict[Any, list[Any]] = {} #diccionario que contiene
        #como claves los vertices del grafo y
        #como valor la lista de vértices a los
        #cuales se puede llegar con una arista saliente
    def grado_salida(self, v: Any) -> int:
        ''' Recibe un vertice y devuelve el número de aristas
        que salen de él'''
        pass

    def grado_entrada(self, v: Any) -> int:
        ''' Recibe un vertice y devuelve el número de aristas
        que llegan a él'''
        pass

    def vertices_siguientes(self, v: Any) -> list[Any]:
        ''' Recibe un vertice y devuelve una lista de todos los vertices
        a los cuales se puede llegar a través de una arista saliente'''
        pass

```

Implemente los métodos `grado_salida`, `grado_entrada` y `vertices_siguientes` de esta clase `GrafoDirigido2`.

- c. Implemente ahora una función `transformarGrafo` que reciba un objeto de tipo `GrafoDirigido1` y devuelva otro objeto equivalente de tipo `GrafoDirigido2`.

```

def transformarGrafo(g: GrafoDirigido1) -> GrafoDirigido2:
    pass

```

3. Encuentre el *ae*-camino de longitud mínima en el siguiente grafo ponderado, utilizando el algoritmo de Dijkstra. Exhibir el camino y la longitud del mismo. Describir de manera detallada los pasos del algoritmo en formato de tabla.

