- **Ejercicio 1:**
  Un documento breve explicando el funcionamiento de la implementación, el uso de estructuras de datos y los resultados obtenidos en un ejemplo de prueba.

Como se puedxse apreciar en la especificación de la función, el código inicia verificando que el grafo tenga la estructura adecuada para poder implementar Dijkstra, como lo serían que el grafo sea un diccionario; que las keys no tengan valores negativos, y que sean números enteros; que el valor asociado a cada key sea una lista de tuplas; y que cada arista sea una tupla con dos elementos, el peso y el nodo que conecta la arista; que no existan dos aristas que conecten a un mismo par de nodos. También se valida la existencia del nodo inicial como una clave en el diccionario. Si alguna de las anteriores no se cumple, se mostrará ValueError, y no se podrá ejecutar.
El código verifica que el grafo no sea inconexo para que se pueda implementar Dijkstra, esto usando búsqueda en amplitud (BFS), si la cantidad de los nodos visitados es igual a los nodos existentes en el grafo, entonces el grafo es conexo.

La función Dijkstra recibe como argumentos 'grafo' y 'nodo_inicio', luego de verificar que el grafo sea válido y conexo, se inicializan las distancias con la ayuda de un diccionario en el cual se ingresará la distancia más corta conocida, estas distancias comenzarán siendo infinito para todos los nodos, a excepción del nodo inicial que es cero, y cambiará su valor conforme se encuentre un valor menor. Igualmente se inicializa una cola de prioridad, la cual contendrá a todos los nodos y sus distancias, aquí utilizamos heap para la cola de prioridad de Dijkstra. Se implementa heap para obtener el nodo con menor distancia en la cola de prioridad. También se establece el conjunto visitados, el cual agregará a cada nodo que ya haya sido procesado.
Cuando ya se ha ejecutado todo lo anterior, se retornará un diccionario que contiene las distancias más cortas obtenidas luego de usar Dijkstra
Dentro del paquete tests, se encontrarán distintas pruebas.
Por ejemplo, para el siguiente grafo, el cual cumple con todas las condiciones para implementar dijsktra anteriores:

```
# Caso extraído del unit test
grafo = {
    1: [(2, 1), (3, 4)],
    2: [(3, 2), (4, 5)],
    3: [(4, 1)],
    4: []
}
print(Dijkstra(grafo, 1))
    ➢  1: 0, 2: 1, 3: 3, 4: 4}
```

- Ejercicio 2:
  Ejemplos de prueba con diferentes grafos y un documento que explique la lógica detrás de la detección de ciclos y el manejo de la recursión.

Como se puede observar en las especificaciones, la función utiliza búsqueda en profundidad DFS para la detección de ciclos, esto con el propósito de, si encuentra algún nodo que ya haya sido visitado durante el proceso, a excepción del nodo padre, poder indicar que existe un ciclo. Se establece un conjunto de nodos visitados, y la función itera sobre los nodos del grafo, su el nodo no ha sido visitado, se realiza el recorrido DFS desde ese nodo. Se establece una pila con lista de tuplas. Si se encuentra un vecino que ya esté en el conjunto de los nodos visitados (distinto al nodo padre), entonces se dice que se ha encontrado un ciclo y se retorna True. De lo contrario, si el DFS termina sin encontrar ciclos, se retornará False.
La recursión se puede ver reflejada en el uso de la pila.

Los ejemplos se encuentran en los test usados

- Ejercicio 3:
  Documento que explique el funcionamiento del algoritmo de Kruskal y Union-Find, acompañado de ejemplos y una prueba de validación.

  Union Find nos permite gestionar conjuntos disjuntos de elementos. El método find se utilizará para encontrar la raíz del conjunto de x. Si un nodo pertenece a un conjunto, entonces todos los nodos intermedios se conectan a este; mejorando así la eficiencia de búsqueda.
  La recursión se da ya que, si parent es distinto de x, x no es su propio representante, así se realiza una llamada recursiva con el fin de encontrar el representante del nodo que pertenece x. Durante este proceso, hay compresión de caminos, mejorando así la velocidad de búsqueda.

  Para el método unión, se combinan dos conjuntos (asegurando que no existan ciclos). Esta se realiza a través de la unión por rango; el árbol del conjunto más pequeño, ahora se convertirá en el hijo del árbol del conjunto más grande. Primero, se verifica que los nodos no estén en el mismo conjunt, si lo están, no se realiza la unión. Si se encuentran en conjuntos diferentes, se hace del nodo de mayor rango el padre del nodo de menor rango. Si los dos tienen el mismo rango, entonces cualquiera de los dos se convierte en el hijo del otro.

  El algoritmo de Kruskal es usado para hallar el AGMA, primero realiza pruebas de validación. Para construir el grafo y verificar la conectividad, se inicial obteniendo el número de nodos distintos a partir de las aristas, se extraen y se almacenan en el conjunto nodos. SI el número de aristas es menor que n-1, el grafo no es conexo (ValueError)
  A través de Union Find, se crea un on¿bjeto con n nodos, se ordenan las aristas por su peso de menor a mayor, y luego se procesan las aristas con menor peso, asegurándose de no formar un ciclo. Para cada arista se hace

la unión de nodos; si estos pertenecen a conjuntos distintos (sin ciclo), la arista se agrega con Union de UnionFind.
Para finalizar, se verifica la conectividad revisando que el AGM tenga n-1 aristas, y se retorna las aristas que retornan el AGM

Las pruebas de validación están en el código directamente, en la forma de unit testing.