6. MergeSort

Descripción: MergeSort es un algoritmo de ordenación basado en la técnica de "divide y vencerás" que divide la lista en sublistas más pequeñas y las ordena recursivamente.

```
def mergesort(arr):
    if len(arr) > 1:  # Si la lista tiene más de un elemento
        mid = len(arr) // 2  # Encuentra el punto medio de la lista
        left = arr[:mid]  # Crea la sublista izquierda
        right = arr[mid:]  # Crea la sublista derecha

        mergesort(left)  # Ordena recursivamente la sublista izquierda
        mergesort(right)  # Ordena recursivamente la sublista derecha

        i = j = k = 0  # Índices para las sublistas y la lista original
        while i < len(left) and j < len(right):  # Mientras haya elementos en ambas sublistas
            if left[i] < right[j]:  # Si el elemento de la izquierda es menor
                arr[k] = left[i]  # Lo coloca en la lista original
                i += 1
            else:
                arr[k] = right[j]  # Si no, coloca el elemento de la derecha
                j += 1
            k += 1

        while i < len(left):  # Si aún hay elementos en la sublista izquierda
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):  # Si aún hay elementos en la sublista derecha
            arr[k] = right[j]
            j += 1
            k += 1
    return arr  # Devuelve la lista ordenada
```

Explicación línea por línea:

- **mid = len(arr) // 2**: Calcula el punto medio de la lista para dividirla en dos sublistas.
- **mergesort(left), mergesort(right)**: Ordena recursivamente las sublistas izquierda y derecha.

- **while i < len(left) and j < len(right)**: Compara los elementos de las sublistas para construir la lista ordenada.

7. Floyd-Warshall

Descripción: El algoritmo de Floyd-Warshall es una solución para encontrar las distancias más cortas entre todos los pares de nodos en un grafo.

```
def floyd_warshall(grafo):
    n = len(grafo)  # Número de nodos en el grafo
    dist = [[float('inf')] * n for _ in range(n)]  # Inicializa la matriz de distancias con inf

    for i in range(n):  # La distancia de un nodo a sí mismo es 0
        dist[i][i] = 0
    for i in range(n):  # Llena la matriz con las distancias directas entre nodos
        for j in range(n):
            if grafo[i][j] != 0:  # Si hay una conexión entre los nodos
                dist[i][j] = grafo[i][j]

    for k in range(n):  # Se iteran todos los nodos intermedios
        for i in range(n):  # Se recorren todos los nodos de inicio
            for j in range(n):  # Se recorren todos los nodos de destino
                if dist[i][j] > dist[i][k] + dist[k][j]:  # Si se encuentra un camino más corto
                    dist[i][j] = dist[i][k] + dist[k][j]  # Actualiza la distancia
    return dist  # Devuelve la matriz de distancias más cortas
```

Explicación línea por línea:

- **dist[i][i] = 0**: La distancia de un nodo a sí mismo es siempre 0.
- **dist[i][j] = grafo[i][j]**: Si hay una arista entre los nodos `i` y `j`, se establece la distancia correspondiente.
- **if dist[i][j] > dist[i][k] + dist[k][j]**: Si el camino pasando por el nodo `k` es más corto que el camino directo, se actualiza la distancia.

8. Bellman-Ford

Descripción: Bellman-Ford es un algoritmo para encontrar las distancias más cortas desde un nodo de inicio a todos los demás nodos en un grafo, incluso con aristas de peso negativo.

```
def bellman_ford(grafo, inicio):
    n = len(grafo)
    dist = [float('inf')] * n  # Inicializa las distancias a infinito
    dist[inicio] = 0  # La distancia al nodo de inicio es 0

    for _ in range(n - 1):  # Repite el proceso n-1 veces
        for i in range(n):  # Recorre todos los nodos
            for vecino, peso in grafo[i].items():  # Para cada vecino y su peso
                if dist[i] + peso < dist[vecino]:  # Si se encuentra una distancia más corta
                    dist[vecino] = dist[i] + peso  # Actualiza la distancia al vecino
    return dist  # Devuelve las distancias más cortas desde el nodo inicial
```

Explicación línea por línea:

- **dist[inicio] = 0**: Se establece la distancia al nodo de inicio como 0.
- **for _ in range(n - 1)**: El algoritmo realiza `n-1` iteraciones, donde `n` es el número de nodos en el grafo.
- **if dist[i] + peso < dist[vecino]**: Si la distancia al vecino a través de un nodo `i` es más corta, se actualiza la distancia.

9. A* (A-star)

Descripción: A* es un algoritmo de búsqueda que encuentra el camino más corto en un grafo, utilizando una heurística para mejorar la eficiencia.

```
import heapq

def a_star(grafo, inicio, fin, heuristica):
    open_list = [(0 + heuristica[inicio], inicio)]  # Cola de prioridad con el valor f(n)
    g_costs = {inicio: 0}  # Costos g para cada nodo
    came_from = {}  # Diccionario que guarda el camino más corto

    while open_list:
        _, nodo_actual = heapq.heappop(open_list)  # Extrae el nodo con el menor valor f(n)

        if nodo_actual == fin:  # Si se ha llegado al nodo final
            path = []
            while nodo_actual in came_from:  # Sigue el camino desde el nodo final
                path.append(nodo_actual)
                nodo_actual = came_from[nodo_actual]
            return path[::-1]  # Devuelve el camino en orden correcto
```

```
        for vecino, peso in grafo[nodo_actual].items():  # Recorre los vecinos
            g_cost = g_costs[nodo_actual] + peso  # Calcula el costo g del vecino
            if vecino not in g_costs or g_cost < g_costs[vecino]:
                g_costs[vecino] = g_cost  # Actualiza el costo g
                f_cost = g_cost + heuristica[vecino]  # Calcula el valor f
                heapq.heappush(open_list, (f_cost, vecino))  # Añade el vecino a la lista abier
                came_from[vecino] = nodo_actual  # Guarda el nodo actual como el predecesor del
    return None  # Si no se encuentra un camino, retorna None
```

Explicación línea por línea:

- **open_list = [(0 + heuristica[inicio], inicio)]**: La lista abierta contiene tuplas con el costo total estimado y el nodo.
- **heapq.heappop(open_list)**: Extrae el nodo con el costo más bajo.
- **came_from[vecino] = nodo_actual**: Guarda el nodo actual como el predecesor del vecino.

10. QuickSort

Descripción: QuickSort es un algoritmo de ordenación eficiente basado en el principio de "divide y vencerás", que selecciona un pivote y reorganiza los elementos alrededor de él.

```
def quicksort(arr):
    if len(arr) <= 1:  # Si la lista tiene 1 o 0 elementos, ya está ordenada
        return arr
    pivot = arr[len(arr) // 2]  # Elige el pivote como el elemento del medio
    left = [x for x in arr if x < pivot]  # Elementos menores que el pivote
    middle = [x for x in arr if x == pivot]  # Elementos iguales al pivote
    right = [x for x in arr if x > pivot]  # Elementos mayores que el pivote
    return quicksort(left) + middle + quicksort(right)  # Ordena recursivamente las sublistas
```

Explicación línea por línea:

- **pivot = arr[len(arr) // 2]**: El pivote es el elemento del medio de la lista.
- **left = [x for x in arr if x < pivot]**: Crea una lista con los elementos menores que el pivote.

- **return quicksort(left) + middle + quicksort(right)**: Ordena las sublistas y las combina para obtener la lista ordenada.