1. Búsqueda Binaria

Descripción: La búsqueda binaria es un algoritmo eficiente para encontrar un valor en una lista ordenada.

```
def busqueda_binaria(arr, x):
    inicio, fin = 0, len(arr) - 1  # Inicializa los índices de inicio y fin para la búsqueda
    while inicio <= fin:  # Mientras el rango de búsqueda sea válido
        medio = (inicio + fin) // 2  # Calcula el índice medio
        if arr[medio] == x:  # Si el elemento en el medio es el que buscamos
            return medio  # Retorna la posición del elemento encontrado
        elif arr[medio] < x:  # Si el valor en el medio es menor que el valor a buscar
            inicio = medio + 1  # Cambia el índice de inicio al siguiente al medio
        else:  # Si el valor en el medio es mayor que el valor a buscar
            fin = medio - 1  # Cambia el índice de fin al anterior al medio
    return -1  # Si no se encuentra el valor, retorna -1
```

Explicación línea por línea:

- **inicio, fin = 0, len(arr) - 1**: Se inicializan los índices de inicio y fin para el rango de búsqueda.
- **medio = (inicio + fin) // 2**: Se calcula el índice medio de la lista.
- **arr[medio] == x**: Se compara el valor en el medio con el valor a buscar.
- **inicio = medio + 1** y **fin = medio - 1**: Si el valor a buscar no está en el medio, se ajustan los límites de búsqueda para continuar buscando en la mitad correspondiente.
- **return -1**: Si no se encuentra el valor, se retorna -1 indicando que el valor no está en la lista.

2. QuickSort

Descripción: QuickSort es un algoritmo de ordenación que utiliza la técnica de "divide y vencerás".

```
def quicksort(arr):
    if len(arr) <= 1:  # Si la lista tiene 0 o 1 elementos, ya está ordenada
        return arr  # Devuelve la lista tal como está
    pivote = arr[len(arr) // 2]  # Selecciona el elemento del medio como pivote
    izquierda = [x for x in arr if x < pivote]  # Sublista de elementos menores que el pivote
```

```
    centro = [x for x in arr if x == pivote]  # Sublista de elementos iguales al pivote
    derecha = [x for x in arr if x > pivote]  # Sublista de elementos mayores que el pivote
    return quicksort(izquierda) + centro + quicksort(derecha)  # Recursivamente ordena y concat
```

Explicación línea por línea:

- **if len(arr) <= 1:** Si la lista tiene un solo elemento o está vacía, no necesita ordenarse.
- **pivote = arr[len(arr) // 2]**: Se selecciona el pivote como el elemento medio de la lista.
- **izquierda, centro, derecha:** Se crean tres sublistas: una con los elementos menores, otra con los elementos iguales y otra con los elementos mayores que el pivote.
- **quicksort(izquierda) + centro + quicksort(derecha):** Recursivamente ordena las sublistas y las concatena.

3. BFS (Breadth-First Search)

Descripción: BFS es un algoritmo de búsqueda que explora los nodos de un grafo nivel por nivel.

```
from collections import deque  # Importa deque para una cola eficiente

def bfs(grafo, inicio):
    visitados = set()  # Un conjunto para llevar el seguimiento de los nodos visitados
    cola = deque([inicio])  # Cola de nodos por visitar, comenzando con el nodo inicial
    while cola:  # Mientras haya nodos en la cola
        nodo = cola.popleft()  # Extrae el primer nodo de la cola
        if nodo not in visitados:  # Si el nodo no ha sido visitado
            visitados.add(nodo)  # Marca el nodo como visitado
            cola.extend(grafo[nodo] - visitados)  # Añade los vecinos no visitados a la cola
    return visitados  # Retorna el conjunto de nodos visitados
```

Explicación línea por línea:

- **deque([inicio])**: Se utiliza una cola para almacenar los nodos que se van a visitar.
- **cola.popleft()**: Extrae el primer nodo de la cola para procesarlo.
- **cola.extend(grafo[nodo] - visitados)**: Añade los nodos vecinos del nodo actual a la cola, si no han sido visitados.
- **return visitados**: Devuelve el conjunto de nodos que han sido visitados.

4. Dijkstra

Descripción: Dijkstra es un algoritmo de búsqueda de caminos más cortos en un grafo ponderado.

```
import heapq  # Importa el módulo para usar la cola de prioridad (heap)

def dijkstra(grafo, inicio):
    distancias = {nodo: float('infinity') for nodo in grafo}  # Inicializa las distancias a inf
    distancias[inicio] = 0  # La distancia al nodo inicial es 0
    prioridad = [(0, inicio)]  # Cola de prioridad que guarda pares (distancia, nodo)

    while prioridad:  # Mientras haya nodos en la cola de prioridad
        distancia_actual, nodo_actual = heapq.heappop(prioridad)  # Extrae el nodo con la menor
        for vecino, peso in grafo[nodo_actual].items():  # Para cada vecino del nodo actual
            distancia = distancia_actual + peso  # Calcula la nueva distancia al vecino
            if distancia < distancias[vecino]:  # Si la nueva distancia es menor
                distancias[vecino] = distancia  # Actualiza la distancia al vecino
                heapq.heappush(prioridad, (distancia, vecino))  # Añade el vecino a la cola
    return distancias  # Devuelve las distancias mínimas desde el nodo inicial
```

Explicación línea por línea:

- **distancias = {nodo: float('infinity') for nodo in grafo}**: Inicializa todas las distancias a infinito.
- **heapq.heappop(prioridad)**: Extrae el nodo con la menor distancia de la cola de prioridad.
- **heapq.heappush(prioridad, (distancia, vecino))**: Añade un nodo a la cola de prioridad con su distancia actualizada.

5. Knapsack Problem

Descripción: El problema de la mochila busca maximizar el valor de los elementos seleccionados sin exceder la capacidad de la mochila.

```
def knapsack(weights, values, capacity):
    n = len(weights)  # Número de elementos en la mochila
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]  # Tabla para almacenar soluciones parcia

    for i in range(n + 1):  # Recorre todos los elementos
```

```
        for w in range(capacity + 1):  # Recorre todas las capacidades posibles
            if i == 0 or w == 0:  # Si no hay elementos o la capacidad es 0
                dp[i][w] = 0  # El valor máximo es 0
            elif weights[i - 1] <= w:  # Si el peso del elemento cabe en la mochila
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
                # Maximaliza el valor de llevar o no llevar el elemento
            else:
                dp[i][w] = dp[i - 1][w]  # Si no cabe, se conserva el valor sin el elemento
    return dp[n][capacity]  # Retorna el valor máximo alcanzable con la capacidad dada
```

Explicación línea por línea:

- **dp[i][w]**: Almacena el valor máximo alcanzable considerando los primeros `i` elementos y una capacidad `w`.
- **max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])**: Decide si incluir o no el elemento `i`.