def gale\_shapley(men\_preferences, women\_preferences):

n = len(men\_preferences)

free\_men = list(men\_preferences.keys())

engaged = {}

women\_engaged = {woman: None for woman in women\_preferences.keys()}

men\_proposals = {man: 0 for man in men\_preferences.keys()}

while free\_men:

man = free\_men[0]

man\_pref = men\_preferences[man]

woman = man\_pref[men\_proposals[man]]

if women\_engaged[woman] is None:

women\_engaged[woman] = man

engaged[man] = woman

free\_men.remove(man)

else:

current\_partner = women\_engaged[woman]

woman\_pref = women\_preferences[woman]

if woman\_pref.index(man) < woman\_pref.index(current\_partner):

free\_men.append(current\_partner)

free\_men.remove(man)

women\_engaged[woman] = man

engaged[man] = woman

del engaged[current\_partner]

men\_proposals[man] += 1

return engaged

# Example usage

men\_preferences = {

'A': ['V', 'W', 'X'],

'B': ['W', 'V', 'X'],

'C': ['V', 'W', 'X']

}

women\_preferences = {

'V': ['A', 'B', 'C'],

'W': ['B', 'C', 'A'],

'X': ['C', 'A', 'B']

}

stable\_marriages = gale\_shapley(men\_preferences, women\_preferences)

print("Stable Marriages:", stable\_marriages)

OUTPUT  
Stable Marriages: {'A': 'V', 'B': 'W', 'C': 'X'}

**Explanation:**

* men\_preferences and women\_preferences are dictionaries where keys are men and women respectively, and values are their preference lists.
* free\_men is a list of men who are currently free.
* engaged is a dictionary where keys are men and values are their current partners.
* women\_engaged is a dictionary tracking which man each woman is currently engaged to.
* men\_proposals is a dictionary tracking how many proposals each man has made.

**Time Complexity:** The time complexity of the Gale-Shapley algorithm is O(n2)O(n^2)O(n2), where nnn is the number of men (or women). This is because in the worst case, each man will propose to every woman, resulting in nnn proposals per man.

import time

def merge\_sort(arr):

if len(arr) > 1:

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i] < right\_half[j]:

arr[k] = left\_half[i]

i += 1

else:

arr[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

arr[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

arr[k] = right\_half[j]

j += 1

k += 1

def main():

arr = list(map(int, input("Enter the numbers to be sorted, separated by spaces: ").split()))

start\_time = time.time()

merge\_sort(arr)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Sorted array:", arr)

print("Execution time:", execution\_time, "seconds")

if \_\_name\_\_ == "\_\_main\_\_":

main()

OUTPUT

Enter the numbers to be sorted, separated by spaces: 38 27 43 3 9 82 10

Sorted array: [3, 9, 10, 27, 38, 43, 82]

Execution time: 0.000123 seconds

**Explanation:**

* The merge\_sort function implements the merge sort algorithm, which is a divide-and-conquer algorithm.
* The input array is divided into two halves, each half is sorted recursively, and then the sorted halves are merged back together.
* The main function takes user input, sorts the array using merge\_sort, and measures the execution time.

**Time Complexity:**

* Merge sort has a time complexity of O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements in the array. This is because the array is repeatedly divided in half (log⁡n\log nlogn divisions) and each division takes O(n)O(n)O(n) time to merge.

import time

def partition(arr, low, high):

pivot = arr[low]

left = low + 1

right = high

done = False

while not done:

while left <= right and arr[left] <= pivot:

left += 1

while arr[right] >= pivot and right >= left:

right -= 1

if right < left:

done = True

else:

arr[left], arr[right] = arr[right], arr[left]

arr[low], arr[right] = arr[right], arr[low]

return right

def quicksort(arr, low, high):

if low < high:

split\_point = partition(arr, low, high)

quicksort(arr, low, split\_point - 1)

quicksort(arr, split\_point + 1, high)

def main():

arr = list(map(int, input("Enter the numbers to be sorted, separated by spaces: ").split()))

start\_time = time.time()

quicksort(arr, 0, len(arr) - 1)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Sorted array:", arr)

print("Execution time:", execution\_time, "seconds")

if \_\_name\_\_ == "\_\_main\_\_":

main()

OUTPUT   
Enter the numbers to be sorted, separated by spaces: 38 27 43 3 9 82 10

Sorted array: [3, 9, 10, 27, 38, 43, 82]

Execution time: 0.000152 seconds

**Explanation:**

* The partition function selects the first element as the pivot and partitions the array such that elements less than the pivot are on the left and elements greater than the pivot are on the right.
* The quicksort function recursively sorts the partitions.
* The main function takes user input, sorts the array using quicksort, and measures the execution time.

**Time Complexity:**

* The average time complexity of quicksort is O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of elements in the array. However, in the worst case (when the pivot is the smallest or largest element), the time complexity can be O(n2)O(n^2)O(n2).

import heapq

import sys

import time

def dijkstra(graph, start):

priority\_queue = []

heapq.heappush(priority\_queue, (0, start))

distances = {vertex: sys.maxsize for vertex in graph}

distances[start] = 0

shortest\_path = {}

while priority\_queue:

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

if current\_distance > distances[current\_vertex]:

continue

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

shortest\_path[neighbor] = current\_vertex

heapq.heappush(priority\_queue, (distance, neighbor))

return distances, shortest\_path

def print\_solution(distances, shortest\_path, start):

print("Vertex\tDistance from Source\tPath")

for vertex in distances:

path = []

current\_vertex = vertex

while current\_vertex != start:

path.insert(0, current\_vertex)

current\_vertex = shortest\_path.get(current\_vertex, start)

path.insert(0, start)

print(f"{vertex}\t{distances[vertex]}\t\t\t{' -> '.join(path)}")

def main():

graph = {}

# Taking input for the graph

num\_vertices = int(input("Enter the number of vertices: "))

for \_ in range(num\_vertices):

vertex = input("Enter vertex: ")

edges = {}

num\_neighbors = int(input(f"Enter the number of neighbors for vertex {vertex}: "))

for \_ in range(num\_neighbors):

neighbor, weight = input(f"Enter neighbor and weight (e.g., B 1): ").split()

edges[neighbor] = int(weight)

graph[vertex] = edges

start = input("Enter the starting location: ")

start\_time = time.time()

distances, shortest\_path = dijkstra(graph, start)

end\_time = time.time()

execution\_time = end\_time - start\_time

print\_solution(distances, shortest\_path, start)

print("Execution time:", execution\_time, "seconds")

if \_\_name\_\_ == "\_\_main\_\_":

main()

OUTPUT

Enter the starting location: A

Vertex Distance from Source Path

A 0 A

B 1 A -> B

C 3 A -> B -> C

D 4 A -> B -> C -> D

Execution time: 0.000215 seconds

**Explanation:**

* The dijkstra function implements Dijkstra’s algorithm using a priority queue to efficiently find the shortest path.
* The print\_solution function displays the shortest distances from the start location to all other locations, along with the path taken.
* The main function takes user input for the starting location, runs the Dijkstra’s algorithm, and measures the execution time.

**Time Complexity:**

* The time complexity of Dijkstra’s algorithm using a priority queue (binary heap) is O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges.

import time

class DisjointSet:

def \_\_init\_\_(self, vertices):

self.parent = {vertex: vertex for vertex in vertices}

self.rank = {vertex: 0 for vertex in vertices}

def find(self, vertex):

if self.parent[vertex] != vertex:

self.parent[vertex] = self.find(self.parent[vertex])

return self.parent[vertex]

def union(self, root1, root2):

if self.rank[root1] > self.rank[root2]:

self.parent[root2] = root1

elif self.rank[root1] < self.rank[root2]:

self.parent[root1] = root2

else:

self.parent[root2] = root1

self.rank[root1] += 1

def kruskal(graph):

edges = [(weight, u, v) for u, adjacent in graph.items() for v, weight in adjacent.items()]

edges.sort()

vertices = list(graph.keys())

disjoint\_set = DisjointSet(vertices)

mst = []

for weight, u, v in edges:

root1 = disjoint\_set.find(u)

root2 = disjoint\_set.find(v)

if root1 != root2:

mst.append((u, v, weight))

disjoint\_set.union(root1, root2)

return mst

def main():

# Prompt user for graph input

graph = {}

num\_vertices = int(input("Enter the number of vertices: "))

for i in range(num\_vertices):

vertex = input(f"Enter vertex {i + 1}: ")

graph[vertex] = {}

num\_neighbors = int(input(f"Enter the number of neighbors for vertex {vertex}: "))

for \_ in range(num\_neighbors):

neighbor, weight = input(f"Enter neighbor and weight (e.g., B 1): ").split()

graph[vertex][neighbor] = int(weight)

start\_time = time.time()

mst = kruskal(graph)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Edges in the minimum spanning tree:")

for u, v, weight in mst:

print(f"{u} - {v}: {weight}")

print("Execution time:", execution\_time, "seconds")

if \_\_name\_\_ == "\_\_main\_\_":

main()

OUTPUT

Edges in the minimum spanning tree:

A - B: 1

C - D: 1

B - C: 2

Execution time: 0.000315 seconds

**Explanation:**

* The DisjointSet class implements a disjoint-set data structure (union-find) to keep track of which vertices are in which components.
* The kruskal function implements Kruskal's algorithm to find the minimum spanning tree (MST). It sorts all edges by weight and adds the smallest edge to the MST if it doesn't form a cycle.
* The main function constructs a sample graph, runs Kruskal's algorithm, and measures the execution time.

**Time Complexity:**

* The time complexity of Kruskal's algorithm is O(Elog⁡E+Elog⁡V)O(E \log E + E \log V)O(ElogE+ElogV), where EEE is the number of edges and VVV is the number of vertices. The sorting of edges takes O(Elog⁡E)O(E \log E)O(ElogE) time, and the union-find operations take O(Elog⁡V)O(E \log V)O(ElogV) time.

import time

def find\_previous\_intervals(requests):

n = len(requests)

p = [0] \* n

for j in range(n):

for i in range(j-1, -1, -1):

if requests[i][1] <= requests[j][0]:

p[j] = i + 1

break

return p

def weighted\_interval\_scheduling(requests):

n = len(requests)

requests = sorted(requests, key=lambda x: x[1]) # Sort by finish time

p = find\_previous\_intervals(requests)

dp = [0] \* (n + 1)

for j in range(1, n + 1):

dp[j] = max(requests[j-1][2] + dp[p[j-1]], dp[j-1])

return dp[n]

def main():

# Prompt user for input

requests = []

num\_requests = int(input("Enter the number of requests: "))

for i in range(num\_requests):

start\_time = int(input(f"Enter start time for request {i + 1}: "))

finish\_time = int(input(f"Enter finish time for request {i + 1}: "))

profit = int(input(f"Enter profit for request {i + 1}: "))

requests.append((start\_time, finish\_time, profit))

start\_time = time.time()

max\_profit = weighted\_interval\_scheduling(requests)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Maximum profit:", max\_profit)

print("Execution time:", execution\_time, "seconds")

if \_\_name\_\_ == "\_\_main\_\_":

main()

OUTPUT

Maximum profit: 13

Execution time: 0.000118 seconds

**Explanation:**

* The find\_previous\_intervals function finds the latest non-overlapping interval for each interval.
* The weighted\_interval\_scheduling function implements the dynamic programming approach to find the maximum profit. It uses an array dp where dp[j] stores the maximum profit for the first j intervals.
* The main function defines a list of requests (each request is a tuple containing start time, finish time, and profit), runs the weighted interval scheduling algorithm, and measures the execution time.

**Time Complexity:**

* The time complexity of the weighted interval scheduling algorithm is O(nlog⁡n)O(n \log n)O(nlogn) due to the sorting step, where nnn is the number of requests. The dynamic programming step takes O(n)O(n)O(n) time.

import time

def is\_subset\_sum(arr, n, sum):

subset = [[False for \_ in range(sum + 1)] for \_ in range(n + 1)]

for i in range(n + 1):

subset[i][0] = True

for i in range(1, n + 1):

for j in range(1, sum + 1):

if j < arr[i - 1]:

subset[i][j] = subset[i - 1][j]

else:

subset[i][j] = subset[i - 1][j] or subset[i - 1][j - arr[i - 1]]

return subset[n][sum]

def main():

arr = list(map(int, input("Enter the set of non-negative integers, separated by spaces: ").split()))

sum\_value = int(input("Enter the sum value: "))

start\_time = time.time()

result = is\_subset\_sum(arr, len(arr), sum\_value)

end\_time = time.time()

execution\_time = end\_time - start\_time

if result:

print(f"A subset with the given sum {sum\_value} exists.")

else:

print(f"No subset with the given sum {sum\_value} exists.")

print("Execution time:", execution\_time, "seconds")

if \_\_name\_\_ == "\_\_main\_\_":

main()

OUTPUT

Enter the set of non-negative integers, separated by spaces: 3 34 4 12 5 2

Enter the sum value: 9

A subset with the given sum 9 exists.

Execution time: 0.000125 seconds

**Explanation:**

* The is\_subset\_sum function uses dynamic programming to determine if there is a subset of arr with a sum equal to the given sum. It constructs a 2D list subset where subset[i][j] is True if there is a subset of the first i elements with a sum equal to j.
* The main function takes user input for the set of integers and the sum value, runs the is\_subset\_sum function, and measures the execution time.

**Time Complexity:**

* The time complexity of the subset sum algorithm is O(n×sum)O(n \times \text{sum})O(n×sum), where nnn is the number of elements in the array and sum\text{sum}sum is the target sum.

def knapsack(weights, values, capacity):

n = len(weights)

dp = [[0] \* (capacity + 1) for \_ in range(n + 1)]

for i in range(1, n + 1):

for w in range(1, capacity + 1):

if weights[i - 1] <= w:

dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])

else:

dp[i][w] = dp[i - 1][w]

max\_value = dp[n][capacity]

selected\_items = []

w = capacity

for i in range(n, 0, -1):

if dp[i][w] != dp[i - 1][w]:

selected\_items.append(i - 1)

w -= weights[i - 1]

selected\_items.reverse()

return max\_value, selected\_items

def main():

# User input for weights

weights = list(map(int, input("Enter the weights of items, separated by spaces: ").split()))

# User input for values

values = list(map(int, input("Enter the values of items, separated by spaces: ").split()))

# User input for capacity

capacity = int(input("Enter the knapsack capacity: "))

# Calculate maximum value and selected items

max\_value, selected\_items = knapsack(weights, values, capacity)

# Output results

print(f"Maximum value that can be carried: {max\_value}")

print(f"Selected items: {selected\_items}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

 **Input Example**:

* weights = [2, 3, 4, 5] : Weight of each item.
* values = [3, 4, 5, 6] : Value of each item.
* capacity = 7 : Maximum capacity of the knapsack.

 **Output**:

* Maximum value that can be carried: 9
* Selected items: [1, 3] : Items at indices 1 (3 value) and 3 (6 value) are selected.

 The nested loops iterate through all items (n iterations) and for each item, potentially iterate through all possible weights (W iterations).

 Therefore, the time complexity is **O(n \* W)**.

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append([u, v, w])

def bellman\_ford(self, src):

# Step 1: Initialize distances from src to all other vertices as INFINITE

distance = [float('inf')] \* self.V

distance[src] = 0

# Step 2: Relax all edges |V| - 1 times.

for \_ in range(self.V - 1):

for u, v, w in self.graph:

if distance[u] != float('inf') and distance[u] + w < distance[v]:

distance[v] = distance[u] + w

# Step 3: Check for negative-weight cycles.

for u, v, w in self.graph:

if distance[u] != float('inf') and distance[u] + w < distance[v]:

print("Graph contains negative weight cycle")

return

# No negative weight cycle found, print distances

print("Vertex Distance from Source")

for i in range(self.V):

print(f"{i}\t\t{distance[i]}")

def main():

num\_vertices = int(input("Enter the number of vertices in the graph: "))

g = Graph(num\_vertices)

num\_edges = int(input("Enter the number of edges: "))

print("Enter edges in the format 'u v w' (from vertex u to vertex v with weight w):")

for \_ in range(num\_edges):

u, v, w = map(int, input().split())

g.add\_edge(u, v, w)

src\_vertex = int(input("Enter the source vertex for Bellman-Ford algorithm: "))

g.bellman\_ford(src\_vertex)

if \_\_name\_\_ == "\_\_main\_\_":

main()

* **Input Example**:
  + The graph is represented as an adjacency list.
  + Example edges are added using add\_edge(u, v, w) where u and v are vertices and w is the weight of the edge.
  + bellman\_ford(0) computes shortest paths from source node 0.
* **Output**:
  + The algorithm prints the shortest distances from the source node to all other nodes.
  + It detects and prints a message if there's a negative weight cycle.

**Time Complexity:**

The time complexity of the Bellman-Ford algorithm is O(V \* E), where V is the number of vertices and E is the number of edges in the graph. This makes it suitable for graphs with negative weight edges and able to handle the detection of negative weight cycles.

def is\_safe(board, row, col, N):

# Check if there is a queen in the current column

for i in range(row):

if board[i][col] == 1:

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check upper diagonal on right side

for i, j in zip(range(row, -1, -1), range(col, N)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, row, N, result):

if row == N:

# Base case: all queens are placed correctly

config = []

for i in range(N):

config.append("".join("Q" if board[i][j] == 1 else "." for j in range(N)))

result.append(config)

return

for col in range(N):

if is\_safe(board, row, col, N):

# Place queen and recursively solve for next row

board[row][col] = 1

solve\_n\_queens\_util(board, row + 1, N, result)

# Backtrack

board[row][col] = 0

def solve\_n\_queens(N):

board = [[0] \* N for \_ in range(N)]

result = []

solve\_n\_queens\_util(board, 0, N, result)

return result

def main():

N = int(input("Enter the number of queens (N): "))

solutions = solve\_n\_queens(N)

print(f"Number of solutions for {N}-Queens problem: {len(solutions)}")

for i, solution in enumerate(solutions, 1):

print(f"Solution {i}:")

for row in solution:

print(row)

print()

if \_\_name\_\_ == "\_\_main\_\_":

main()

 **Input Example**:

* N = 4 indicates a 4x4 chessboard.

 **Output**:

* The program will print all possible solutions where N queens are placed on the board such that no two queens threaten each other.
* Each solution is represented as a list of strings, where "Q" represents a queen and "." represents an empty cell.

 **Implementation Details**:

* is\_safe function checks whether it's safe to place a queen in the current position (row, col).
* solve\_n\_queens\_util is a recursive helper function that attempts to place queens row by row.
* solve\_n\_queens initializes the board and collects all valid configurations (solutions).

 **Complexity**:

* The time complexity of this solution is exponential, O(N!), due to the factorial number of possible configurations to check.
* However, pruning techniques and early termination can significantly reduce the number of configurations explored, making it practical for reasonably small values of N (typically up to 15 or so).

 **Gale-Shapley Algorithm**:

* Implement to find stable marriages between men and women.
* Time complexity typically O(n^2) where n is the number of individuals (men or women).

 **Merge Sort Algorithm**:

* Implement merge sort with user-defined input.
* Time complexity is O(n log n).

 **Quicksort Algorithm**:

* Implement quicksort using the first element as the pivot.
* Time complexity O(n log n) on average, O(n^2) worst-case.

 **Dijkstra's Algorithm**:

* Implement to find the shortest path from a given source to all other locations.
* Time complexity O((V + E) log V) using a priority queue with Fibonacci heap.

 **Kruskal's Algorithm**:

* Implement to find the minimum spanning tree using a greedy approach.
* Time complexity O(E log V) using union-find data structure.

 **Weighted Interval Scheduling Algorithm**:

* Implement using dynamic programming to maximize profit.
* Time complexity O(n log n) using a suitable sorting technique.

 **Subset Sum Problem**:

* Implement to determine if there's a subset with a given sum.
* Time complexity O(n \* sum) using dynamic programming.

 **Knapsack Problem**:

* Implement using dynamic programming to maximize value within weight limit.
* Time complexity O(n \* W) where W is the capacity of the knapsack.

 **Bellman-Ford Algorithm**:

* Implement to find shortest paths from a source to all nodes.
* Time complexity O(V \* E), suitable for graphs with negative weight edges.

 **N-Queens Problem**:

* Implement to find all solutions on a 4x4 chessboard.
* Time complexity O(n!), where n is the size of the chessboard.

