PART B

```
// 1

def gale_shapley(men_preferences, women_preferences):
    n = len(men_preferences)
    engaged = {}
    men_free = set(range(n))

    while men_free:
        m = men_free.pop()
        w = men_preferences[m].pop(0)

        if w not in engaged:
            engaged[w] = m
        else:
            m1 = engaged[w]
            w_prefs = women_preferences[w]
            if w_prefs.index(m) < w_prefs.index(m1):
                engaged[w] = m
                men_free.add(m1)
            else:
                men_free.add(m)

    return engaged

# Example preferences for men and women
men_preferences = {
    0: [1, 0, 2, 3],
    1: [0, 1, 2, 3],
    2: [0, 1, 2, 3],
    3: [1, 0, 2, 3]
}

women_preferences = {
    0: [1, 0, 2, 3],
    1: [0, 1, 2, 3],
    2: [1, 0, 2, 3],
    3: [1, 0, 2, 3]
}

# Perform Gale-Shapley algorithm
stable_matching = gale_shapley(men_preferences, women_preferences)

# Print the stable matching result
for woman, man in stable_matching.items():
    print(f"Man {man} is engaged to Woman {woman}.")
```

Man 0 is engaged to Woman 1.

Man 1 is engaged to Woman 0.

Man 2 is engaged to Woman 2.

Man 3 is engaged to Woman 3.

2.

```
def merge_sort(arr):
    if len(arr) <= 1:
```

PART B

```
        return arr

    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    left = merge_sort(left)
    right = merge_sort(right)

    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result


# Example usage
arr = [8, 4, 2, 1, 6, 9, 3, 5, 7]
sorted_arr = merge_sort(arr)
print("Sorted array:", sorted_arr)
```

original array [8, 4, 2, 1, 6, 9, 3, 5, 7]

Sorted array: [1, 2, 3, 4, 5, 6, 7, 8, 9]

3.

```
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
```

PART B

```
    return distances


# Example graph as an adjacency list
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 3, 'D': 2},
    'C': {'A': 4, 'B': 3, 'D': 5},
    'D': {'B': 2, 'C': 5}
}

start_node = 'A'
shortest_distances = dijkstra(graph, start_node)
print("Shortest distances from node", start_node + ":", shortest_distances)
```

5.

```
def drama_venue_allocation(requests):
    requests.sort(key=lambda x: x[1])  # Sort requests based on finish
times
    prev_profit = curr_profit = 0

    for start_time, finish_time, profit in requests:
        max_profit = max(prev_profit + profit, curr_profit)
        prev_profit, curr_profit = curr_profit, max_profit

    return curr_profit

# Example requests: (start_time, finish_time, profit)
requests = [(1, 2, 100), (2, 5, 200), (3, 6,300 ), (4, 8,
400),(4,9,500),(6,10,100)]

# Calculate the maximum profit using dynamic programming
max_profit = drama_venue_allocation(requests)

print("Maximum Profit:", max_profit)
```

Maximum Profit: 900

6.

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [0] * (capacity + 1)

    for i in range(n):
        for w in range(capacity, weights[i] - 1, -1):
            dp[w] = max(dp[w], values[i] + dp[w - weights[i]])

    return dp[capacity]


# Example values and weights for items
values = [10,4,9,11]
weights = [3,5,6,2]
capacity = 7
```

PART B

```
max_value = knapsack(values, weights, capacity)
print("Maximum value:", max_value)
```

Maximum value: 21

7.

```
def bellman_ford(vertices, edges, start):
    distances = {v: float('inf') for v in vertices}
    distances[start] = 0

    for _ in vertices:
        for u, v, weight in edges:
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight

    return distances


# Example usage
vertices = ['A', 'B', 'C', 'D', 'E','F']
edges = [('A', 'B', -4), ('B', 'E', -2), ('B', 'D', -1), ('D', 'A', 6),
         ('A', 'F', -3), ('D', 'F', 4), ('E', 'F',
2),('C','B',8),('C','F',3)]

start_vertex = 'A'
shortest_distances = bellman_ford(vertices, edges, start_vertex)
print("Shortest distances from vertex", start_vertex + ":",
shortest_distances)
```

Shortest distances from vertex A: {'A': 0, 'B': -4, 'C': inf, 'D': -5, 'E': -6, 'F': -4}

8.