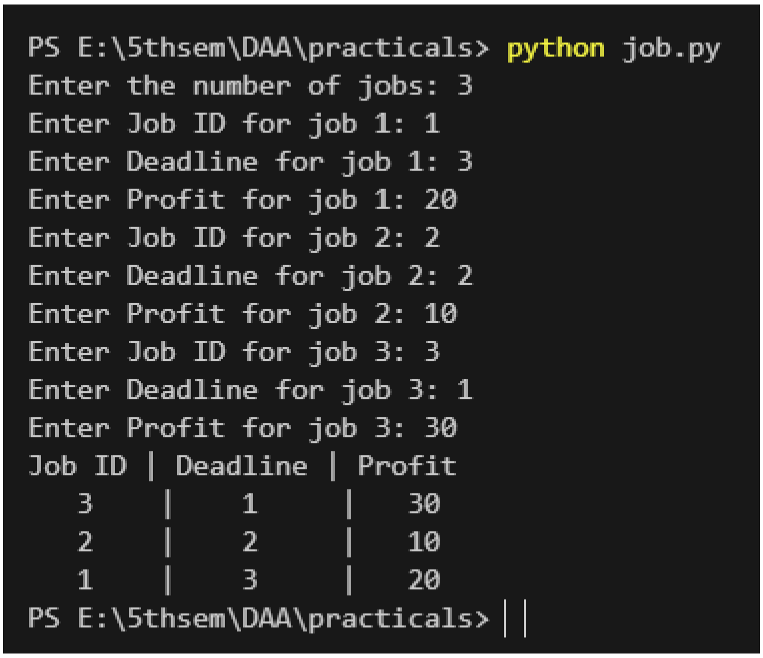
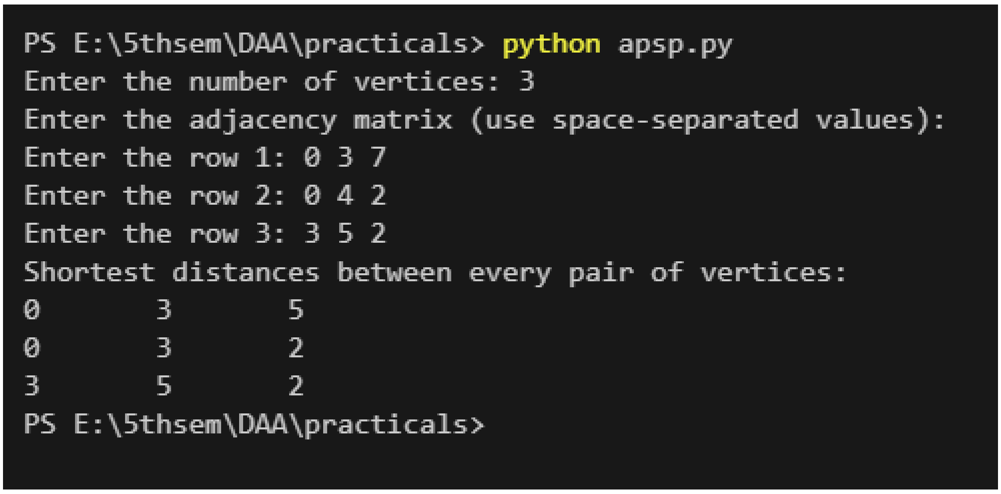
```
class Job:
    def __init__(self, id, deadline, profit):
        self.id = id
        self.deadline = deadline
        self.profit = profit

def job_sequencing(jobs, n):
    # Sort jobs according to descending order of profit
    jobs.sort(key=lambda job: job.profit, reverse=True)

    # Find the maximum deadline
    max_deadline = max(job.deadline for job in jobs)

    # Initialize a slot array to keep track of free time slots
    slots = [-1] * max_deadline

    # Initialize the result array
    result = [None] * n

    # Iterate through all jobs
    for job in jobs:
        # Find a free slot for this job (from the last possible slot)
        for j in range(min(max_deadline, job.deadline) - 1, -1, -1):
            if slots[j] == -1:
                slots[j] = job.id
                result[j] = job
                break

    # Print the jobs in the sequence of their deadlines
    print("Job ID | Deadline | Profit")
    for job in result:
        if job is not None:
            print(f"   {job.id}    |    {job.deadline}     |   {job.profit}")

if __name__ == "__main__":
    # Input number of jobs
    n = int(input("Enter the number of jobs: "))

    # List to store jobs
    jobs = []

    # Input job details
    for i in range(n):
        job_id = int(input(f"Enter Job ID for job {i + 1}: "))
    deadline = int(input(f"Enter Deadline for job {i + 1}: "))
    profit = int(input(f"Enter Profit for job {i + 1}: "))
    jobs.append(Job(job_id, deadline, profit))

# Perform job sequencing
job_sequencing(jobs, n)
```



```
PS E:\5thsem\DAA\practicals> python job.py
Enter the number of jobs: 3
Enter Job ID for job 1: 1
Enter Deadline for job 1: 3
Enter Profit for job 1: 20
Enter Job ID for job 2: 2
Enter Deadline for job 2: 2
Enter Profit for job 2: 10
Enter Job ID for job 3: 3
Enter Deadline for job 3: 1
Enter Profit for job 3: 30
Job ID | Deadline | Profit
   3    |    1     |   30
   2    |    2     |   10
   1    |    3     |   20
PS E:\5thsem\DAA\practicals>
```

```
import sys

def floyd_warshall(graph, n):
    # Initialize the distance matrix
    dist = [[float('inf')] * n for _ in range(n)]

    # Set the distance from each node to itself to 0
    for i in range(n):
        dist[i][i] = 0

    # Set initial distances based on the input graph
    for u in range(n):
        for v in range(n):
            if graph[u][v] != float('inf'):
                dist[u][v] = graph[u][v]

    # Floyd-Warshall algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

def print_solution(dist):
    n = len(dist)
    print("Shortest distances between every pair of vertices:")
    for i in range(n):
        for j in range(n):
            if dist[i][j] == float('inf'):
                print("INF", end="\t")
            else:
                print(dist[i][j], end="\t")
        print()

if __name__ == "__main__":
    # Input number of vertices
    n = int(input("Enter the number of vertices: "))

    # Initialize graph with infinities
    graph = [[float('inf')] * n for _ in range(n)]

    print("Enter the adjacency matrix (use space-separated values):")
```

```
# Input adjacency matrix
for i in range(n):
    row = input(f"Enter the row {i + 1}: ").split()
    for j in range(n):
        value = row[j]
        if value.lower() == 'inf':
            graph[i][j] = float('inf')
        else:
            graph[i][j] = int(value)

# Run Floyd-Warshall algorithm
dist = floyd_warshall(graph, n)

# Print the result
print_solution(dist)
```



```
PS E:\5thsem\DAA\practicals> python apsp.py
Enter the number of vertices: 3
Enter the adjacency matrix (use space-separated values):
Enter the row 1: 0 3 7
Enter the row 2: 0 4 2
Enter the row 3: 3 5 2
Shortest distances between every pair of vertices:
0       3       5
0       3       2
3       5       2
PS E:\5thsem\DAA\practicals>
```