In [4]:

```python
#1. Bellman-Ford Algorithm
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):
        dist = [float("Inf")] * self.V
        dist[src] = 0

        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        self.print_solution(dist)

    def print_solution(self, dist):
        print("Vertex \tDistance from Source")
        for i in range(self.V):
            print(f"{i}\t\t{dist[i]}")

# Example usage
g = Graph(5)
g.add_edge(0, 1, -1)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 3)
g.add_edge(1, 3, 2)
g.add_edge(1, 4, 2)
g.add_edge(3, 2, 5)
g.add_edge(3, 1, 1)
g.add_edge(4, 3, -3)

g.bellman_ford(0)
```

```
Vertex  Distance from Source
0               0
1               -1
2               2
3               -2
4               1
```

In [5]:

```
#2. Warshall's Algorithm
def warshall_algorithm(graph):
    V = len(graph)
    reach = [[0] * V for _ in range(V)]

    for i in range(V):
        for j in range(V):
            reach[i][j] = graph[i][j]

    for k in range(V):
        for i in range(V):
            for j in range(V):
                reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][

    print("Transitive closure matrix is:")
    for i in range(V):
        for j in range(V):
            print(reach[i][j], end=" ")
        print()

# Example usage
graph = [
    [1, 1, 0, 1],
    [0, 1, 1, 0],
    [0, 0, 1, 1],
    [0, 0, 0, 1]
]

warshall_algorithm(graph)
```

```
Transitive closure matrix is:
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
```

In [6]:

```
#3. Coin Change Problem
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] = min(dp[x], dp[x - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

# Example usage
coins = [1, 2, 5]
amount = 11
print(f"Minimum coins needed: {coin_change(coins, amount)}")
```

```
Minimum coins needed: 3
```

In [7]:

```
#4. Knapsack Problem Using Greedy
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
        self.ratio = value / weight

def knapsack_greedy(values, weights, W):
    items = [Item(values[i], weights[i]) for i in range(len(values))]
    items.sort(key=lambda x: x.ratio, reverse=True)

    max_value = 0
    for item in items:
        if W >= item.weight:
            W -= item.weight
            max_value += item.value
        else:
            max_value += item.ratio * W
            break

    return max_value

# Example usage
values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
print(f"Maximum value in Knapsack = {knapsack_greedy(values, weights, W
```

```
Maximum value in Knapsack = 240.0
```

In [8]:

```
#5. Job Sequencing with Deadlines
class Job:
    def __init__(self, id, deadline, profit):
        self.id = id
        self.deadline = deadline
        self.profit = profit

def job_sequencing(jobs):
    jobs.sort(key=lambda x: x.profit, reverse=True)

    n = len(jobs)
    result = [False] * n
    job_sequence = [-1] * n

    for i in range(len(jobs)):
        for j in range(min(n, jobs[i].deadline) - 1, -1, -1):
            if not result[j]:
                result[j] = True
                job_sequence[j] = jobs[i].id
                break

    print("Job sequence to maximize profit:")
    print([job for job in job_sequence if job != -1])

# Example usage
jobs = [Job(1, 2, 100), Job(2, 1, 19), Job(3, 2, 27), Job(4, 1, 25), Jo
job_sequencing(jobs)
```

```
Job sequence to maximize profit:
[3, 1, 5]
```

In [ ]: