1. **Assembly Line Scheduling**

```
def val(n):
    x = []
    for _ in range(n):
        num= int(input("Enter a number: "))
        x.append(num)
    return x

def asl(a1, a2, t1, t2, e1, e2, x1, x2):
    g = list(a1)
    n = len(a1)
    F1 = [0] * n
    F2 = [0] * n
    F1[0] = e1 + a1[0]
    F2[0] = e2 + a2[0]
    for i in range(1, n):
        F1[i] = min(F1[i-1] + a1[i], F2[i-1] + t2[i-1] + a1[i])
        F2[i] = min(F2[i-1] + a2[i], F1[i-1] + t1[i-1] + a2[i])
    f1 = F1[n-1] + x1
    f2 = F2[n-1] + x2
    return min(f1, f2)
a1 = int(input("How many numbers for a1? "))
r = val(a1)
print(r)
a2 = int(input("How many numbers for a2? "))
q = val(a2)
print(q)
t1 = int(input("How many numbers for t1? "))
q1 = val(t1)
print(q1)
t2 = int(input("How many numbers for t2? "))
q2 = val(t2)
print(q2)
e1 = int(input("How many numbers for e1? "))
e2 = int(input("How many numbers for e2? "))
x1 = int(input("How many numbers for x1? "))
x2 = int(input("How many numbers for x2? "))
print(assembly_line(r, q, q1, q2, e1, e2, x1, x2))
```

2. **Knapsack problem and Memory**

```
def knapsack(values, weights, W):
    n = len(values)
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(W + 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]

    return dp[n][W]

values = [60, 100, 120]
weights = [10, 20, 30]
W = 50
print(knapsack(values, weights, W))
```

3. **Warshall's & Floyd's Algorithm**

```
def floyd(graph):
    V = len(graph)
    dist = [[float('inf')] * V for _ in range(V)]
    for i in range(V):
        for j in range(V):
            if i == j:
                dist[i][j] = 0
            elif graph[i][j]:
                dist[i][j] = graph[i][j]
    for k in range(V):
        for i in range(V):
            for j in range(V):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist

graph = [[0, 3, float('inf'), 5],[2, 0, float('inf'), 4],[float('inf'), 1, 0, float('inf')],[float('inf'),
float('inf'), 2, 0]]

shortest = floyd(graph)
for row in shortest:
    print(row)
```

4. **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(self, src):
        distance = [float('Inf')] * self.V
        distance[src] = 0
        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
        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
        print("Vertex Distance from Source")
        for i in range(self.V):
            print(f"{i} \t\t {distance[i]}")


if __name__ == "__main__":
    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(0)
```