1.Computing binomial coefficient

Program:

def binomial\_coefficient(n, k):

if k > n - k: # Take advantage of symmetry

k = n - k

c = 1

for i in range(k):

c = c \* (n - i) // (i + 1)

return c

# Example usage:

n = 5

k = 2

print(binomial\_coefficient(n, k)) # Output: 10

2.bellman ford algorithm

Program:

def bellman\_ford(vertices, edges, src):

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

dist = [float('inf')] \* vertices

dist[src] = 0

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

for \_ in range(vertices - 1):

for u, v, w in edges:

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

dist[v] = dist[u] + w

# Step 3: Check for negative-weight cycles

for u, v, w in edges:

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

print("Graph contains negative weight cycle")

return None

return dist

# Example usage:

vertices = 5

edges = [

(0, 1, -1),

(0, 2, 4),

(1, 2, 3),

(1, 3, 2),

(1, 4, 2),

(3, 2, 5),

(3, 1, 1),

(4, 3, -3)

]

source = 0

distances = bellman\_ford(vertices, edges, source)

if distances is not None:

print("Vertex Distance from Source")

for i in range(vertices):

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

3. Warshal Floyd

Program:

def floyd\_warshall(vertices, edges):

# Initialize distance matrix

dist = [[float('inf')] \* vertices for \_ in range(vertices)]

for i in range(vertices):

dist[i][i] = 0

for u, v, w in edges:

dist[u][v] = w

# Update the distance matrix

for k in range(vertices):

for i in range(vertices):

for j in range(vertices):

if dist[i][j] > dist[i][k] + dist[k][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist

# Example usage:

vertices = 4

edges = [

(0, 1, 5),

(0, 3, 10),

(1, 2, 3),

(2, 3, 1)

]

distances = floyd\_warshall(vertices, edges)

# Print the result

print("Distance matrix:")

for row in distances:

print(row)

4.meet in the middle

Program:

def subset\_sum(nums, target):

# Function to generate all possible subset sums of a list

def generate\_subsets\_sums(nums):

sums = {0}

for num in nums:

new\_sums = {num + s for s in sums}

sums.update(new\_sums)

return sums

# Divide the nums into two halves

mid = len(nums) // 2

left\_part = nums[:mid]

right\_part = nums[mid:]

# Generate all subset sums for both halves

left\_sums = generate\_subsets\_sums(left\_part)

right\_sums = generate\_subsets\_sums(right\_part)

# Check if there's a pair of sums that add up to the target

for s in left\_sums:

if (target - s) in right\_sums:

return True

return False

# Example usage:

nums = [3, 34, 4, 12, 5, 2]

target = 9

print(subset\_sum(nums, target)) # Output: True