

1. COMPUTING BINOMIAL COEFFICIENT

```
def binomial_coefficient(n, k):
    if k > n:
        return 0
    if k == 0 or k == n:
        return 1
    C = [[0 for x in range(k + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j]
    return C[n][k]

n = 5
k = 2
print(f"Binomial Coefficient C({n},{k}) is {binomial_coefficient(n, k)}")
```

2. BELLMAN FORD

```
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 Distance from Source")
    for i in range(self.V):
        print(f"{i}\t\t{dist[i]}")

```

```

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)

```

3. WARSHAL FLOYD

```

def floyd_warshall(graph):
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))

```

```

V = len(graph)
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, 5, float("Inf"), 10],
    [float("Inf"), 0, 3, float("Inf")],
    [float("Inf"), float("Inf"), 0, 1],
    [float("Inf"), float("Inf"), float("Inf"), 0]
]

```

```

distance_matrix = floyd_warshall(graph)
print("Shortest distances between every pair of vertices:")
for row in distance_matrix:
    print(row)

```

4. MEET IN THE MIDDLE TECHNIQUE

```

def meet_in_the_middle(arr, S):
    n = len(arr)
    left = arr[:n//2]
    right = arr[n//2:]

```

```

def subset_sums(arr):
    sums = []
    n = len(arr)
    for i in range(1 << n):
        sum = 0
        for j in range(n):
            if i & (1 << j):

```

```

        sum += arr[j]
    sums.append(sum)
return sums

left_sums = subset_sums(left)
right_sums = subset_sums(right)

right_sums.sort()
for sum in left_sums:
    if binary_search(right_sums, S - sum):
        return True
return False

def binary_search(arr, x):
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == x:
            return True
        elif arr[mid] < x:
            lo = mid + 1
        else:
            hi = mid - 1
    return False

# Example usage
arr = [3, 34, 4, 12, 5, 2]
S = 9
if meet_in_the_middle(arr, S):
    print("Found a subset with the given sum")
else:

```

```
print("No subset with the given sum")
```