Ex. No.: 5a DYNAMIC PROGRAMMING - COIN CHANGE PROBLEM
Date:

AIM:

To write a python program to implement the coin change problem using dynamic programming technique.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Two choices for a coin of a particular denominator either to include or to exclude.
STEP 3: At coins[n-1], we can take as, any instances of that can,
i.e., count(coins ,n, sum-coins [n- i]) then we can move to coins[n-2].
STEP 4: After moving to coins[n-2], we cannot move back and cannot make choices for coins[n-1] i.e., count(coins , n-1 , sum).
STEP 5: Find the total number of ways , to we will add these – possible choices.
i.e., count(coins , n ,sum-coins[n-1])+ count(coins , n-1 ,sum).
STEP 6: Stop the program.

PROGRAM:

```
def count(S, target):
        if target == 0:
                return 1
        if target < 0:
                return 0
        result = 0
        for c in S:
                result += count(S, target - c)
        return result
if __name__== '__main__':
        S = [1, 2, 3]
        target = 4
        print('The total number of ways to get the desired change is ', count(S, target))
```

OUTPUT:

RESULT:

Thus the python program for coin change problem using dynamic programming was implemented and executed successfully.

Ex. No.: 5b **DYNAMIC PROGRAMMING - WARSHALL'S AND**
Date: **FLOYD'S ALGORITHM**

AIM:

To write a python program to implement Warshall's and Floyd's algorithm using dynamic programming technique.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Create a matrix A0 of dimension n*n where n is the number of vertices.
STEP 3: Create a matrix A1 using matrix A0 . The elements in the first column and the first row left as they are (A[i][k]+A[k][j])if (A[i][j]>A[i][k]+A[k][j].
STEP 4: Similarly, A2 is created using A1 and similarly A3 and A4 is also created.
STEP 5: A4 gives the shortest path between each pair of vertices.
STEP 6: Stop the program.

PROGRAM:

```
nV = 4

INF = 999

def floyd_warshall(G):
        distance = list(map(lambda i: list(map(lambda j: j, i)), G))

    for k in range(nV):
        for i in range(nV):
                for j in range(nV):
                        distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
    print_solution(distance)

def print_solution(distance):
   for i in range(nV):
        for j in range(nV):
                if(distance[i][j] == INF):
                        print("INF", end=" ")
                else:
                        print(distance[i][j], end=" ")
        print(" ")

G = [ [0, 3, INF, 5],
        [2, 0, INF, 4],
```

```
        [INF, 1, 0, INF],
        [INF, INF, 2, 0]]
floyd_warshall(G)
```

OUTPUT:

RESULT:

Thus the python program for Warshall's and Floyd's algorithm using dynamic programming technique was implemented and executed successfully.

Ex. No.: 5c DYNAMIC PROGRAMMING - KNAPSACK PROBLEM

Date:

AIM:

To write a python program to implement the Knapsack problem using dynamic programming technique.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Initialize knapSack*(0, j)* = 0 for $j \geq 0$ and knapSack*(i, 0)* = 0 for $i \geq 0$.

STEP 3: Do the following to divide all the subsets of the first *i* items that fit the knapsack of capacity *j* into two categories: those that do not include the *i*th item:

3.1**:** Among the subsets that do not include the *i*th item, the value of an optimal subset is, by definition, knapSack*(i − 1, j)*.

3.2: Among the subsets that do include the *i*th item (hence, $j - wi \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - wi$. The value of such an optimal subset is *vi*+ knapSack*(i − 1, j − wi)*.

STEP 4: Print optimal solution.

STEP 5: Stop the program.

PROGRAM:

```
def knapSack(W, wt, val, n):
    if n == 0 or W == 0:
        return 0
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:
        return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1), knapSack(W, wt, val, n-1))

val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W, wt, val, n)
```

OUTPUT:

RESULT:

Thus the python program for the Knapsack problem using dynamic programming technique was implemented and executed successfully.