NAME: MD. NAYEEM MOLLA

Subject: Algorithm Design and Analysis

Experiment IV: Dynamic Programming

1. Purpose

Apply the dynamic programming algorithm to solve the practical problems and implement it in programming language.

2. Main requirements

(1) Write and debug the traveling salesman problem by using the dynamic programming algorithm.

(2) Write and debug matrix chain multiplication by using the dynamic programming algorithm.

(3) Choose one or two.

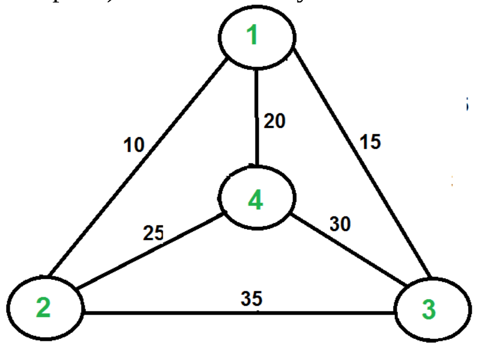
3. Instrument and equipment

PC-compatible (language-free).

4. Algorithm's principles

4.1 Traveling Salesman Problem

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem. Following are different solutions for the traveling salesman problem.

Brute-Force Solution:
1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.
Time Complexity: Θ(n!)

Dynamic Programming:
Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)? To calculate cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.*
We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},

C(S, i) = dist(1, i)

Else if size of S is greater than 2.

C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

For a set of size n, we consider n-2 subsets each of size n-1 such that all subsets don't have nth in them. Using the above recurrence relation, we can write dynamic programming based solution. There are at most O($n*2^n$) sub-problems, and each one takes linear time to solve. The total running time is therefore O(n^2*2^n).

4.2 Matrix Chain Multiplication Problem

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

```
(ABC)D = (AB)(CD) = A(BCD) = ....
```

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

```
(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
```

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain. For example:

```
Input: p[] = {40, 20, 30, 10, 30}
Output: 26000
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}
Output: 30000
There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}
Output: 6000
There are only two matrices of dimensions 10x20 and 20x30. So there
is only one way to multiply the matrices, cost of which is 10*20*30
```

A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

Implementation of the Algorithm Using Python Language:

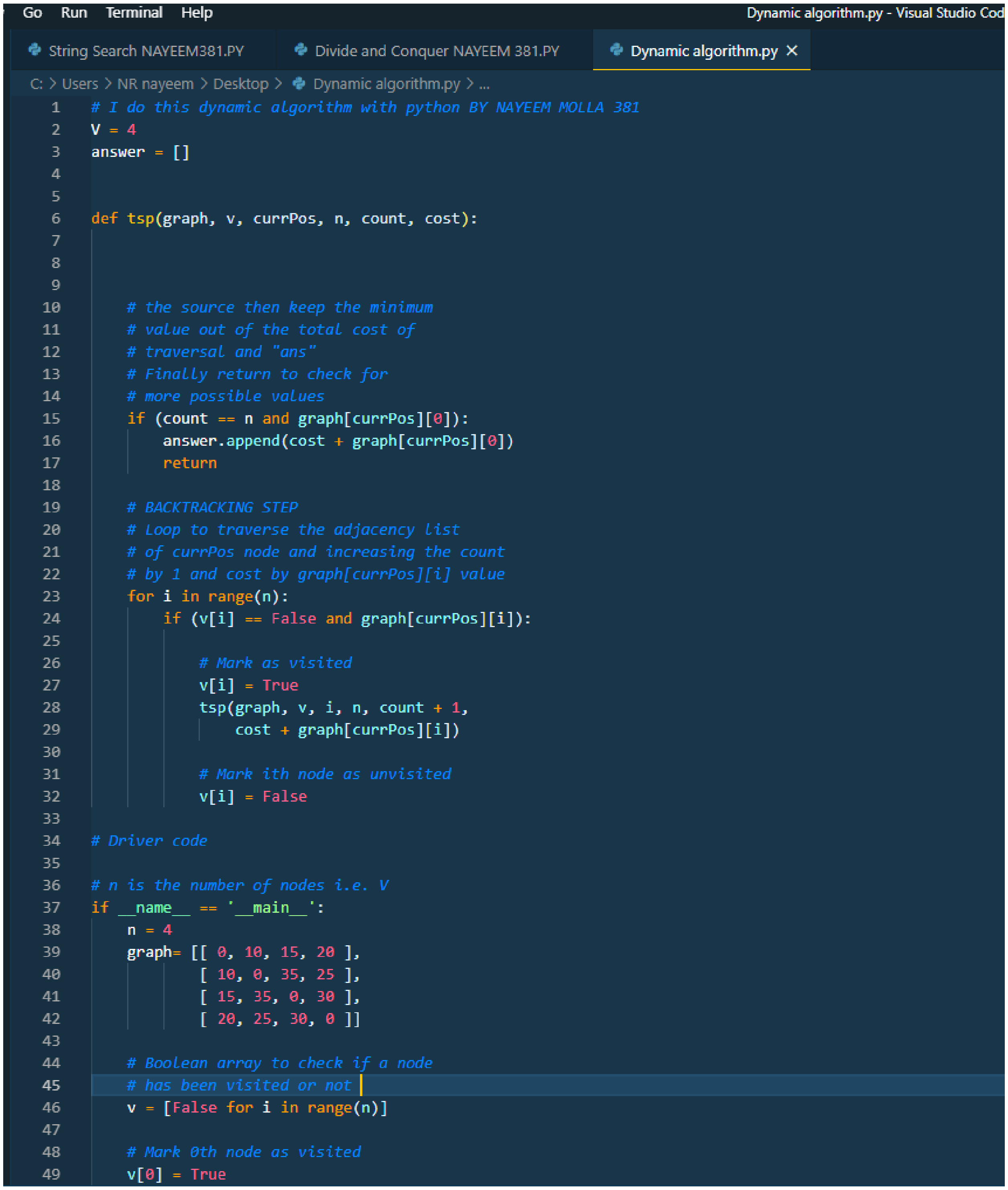
SOURCE CODE:

```
# I do this dynamic algorithm with python BY NAYEEM MOLLA 381

V = 4

answer = []



def tsp(graph, v, currPos, n, count, cost):




  # the source then keep the minimum

  # value out of the total cost of

  # traversal and "ans"

  # Finally return to check for

  # more possible values

  if (count == n and graph[currPos][0]):

    answer.append(cost + graph[currPos][0])

    return


  # BACKTRACKING STEP

  # Loop to traverse the adjacency list

  # of currPos node and increasing the count

  # by 1 and cost by graph[currPos][i] value
```

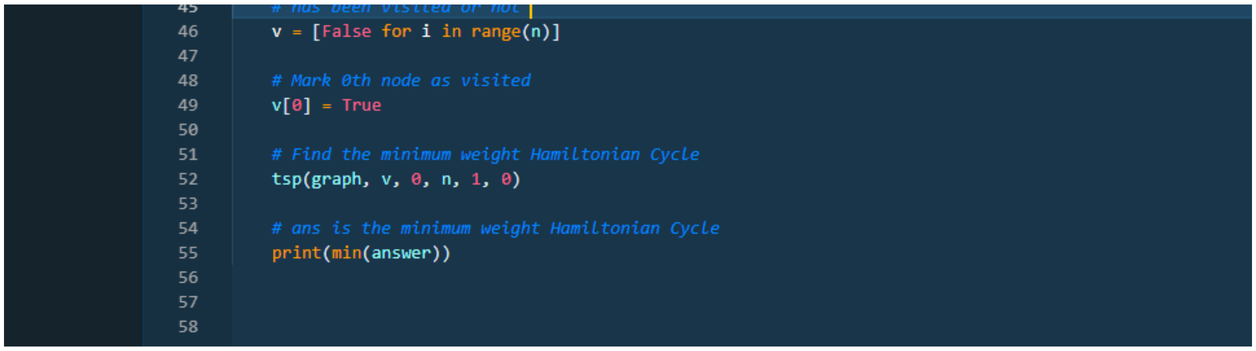
```
    for i in range(n):
        if (v[i] == False and graph[currPos][i]):

            # Mark as visited
            v[i] = True
            tsp(graph, v, i, n, count + 1,
                cost + graph[currPos][i])

            # Mark ith node as unvisited
            v[i] = False

# Driver code

# n is the number of nodes i.e. V
if __name__ == '__main__':
    n = 4
    graph= [[ 0, 10, 15, 20 ],
            [ 10, 0, 35, 25 ],
            [ 15, 35, 0, 30 ],
            [ 20, 25, 30, 0 ]]

    # Boolean array to check if a node
    # has been visited or not
    v = [False for i in range(n)]

    # Mark 0th node as visited
    v[0] = True

    # Find the minimum weight Hamiltonian Cycle
```

```
tsp(graph, v, 0, n, 1, 0)


# ans is the minimum weight Hamiltonian Cycle
print(min(answer))
```

INPUT:

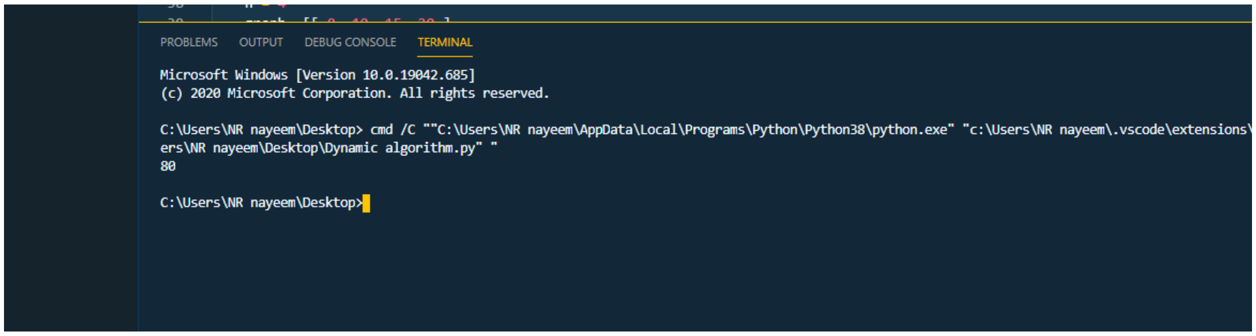

Go Run Terminal Help
Dynamic algorithm.py - Visual Studio Cod
String Search NAYEEM381.PY
Divide and Conquer NAYEEM 381.PY
Dynamic algorithm.py
C: > Users > NR nayeem > Desktop > Dynamic algorithm.py > ...

```python
# I do this dynamic algorithm with python BY NAYEEM MOLLA 381
V = 4
answer = []


def tsp(graph, v, currPos, n, count, cost):



    # the source then keep the minimum
    # value out of the total cost of
    # traversal and "ans"
    # Finally return to check for
    # more possible values
    if (count == n and graph[currPos][0]):
        answer.append(cost + graph[currPos][0])
        return

    # BACKTRACKING STEP
    # Loop to traverse the adjacency list
    # of currPos node and increasing the count
    # by 1 and cost by graph[currPos][i] value
    for i in range(n):
        if (v[i] == False and graph[currPos][i]):

            # Mark as visited
            v[i] = True
            tsp(graph, v, i, n, count + 1,
                cost + graph[currPos][i])

            # Mark ith node as unvisited
            v[i] = False

# Driver code

# n is the number of nodes i.e. V
if __name__ == '__main__':
    n = 4
    graph= [[ 0, 10, 15, 20 ],
            [ 10, 0, 35, 25 ],
            [ 15, 35, 0, 30 ],
            [ 20, 25, 30, 0 ]]

    # Boolean array to check if a node
    # has been visited or not
    v = [False for i in range(n)]

    # Mark 0th node as visited
    v[0] = True
```



```
    # has been visited or not
    v = [False for i in range(n)]

    # Mark 0th node as visited
    v[0] = True

    # Find the minimum weight Hamiltonian Cycle
    tsp(graph, v, 0, n, 1, 0)

    # ans is the minimum weight Hamiltonian Cycle
    print(min(answer))
```

OUTPUT:



```
Microsoft Windows [Version 10.0.19042.685]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\NR nayeem\Desktop> cmd /C ""C:\Users\NR nayeem\AppData\Local\Programs\Python\Python38\python.exe" "c:\Users\NR nayeem\.vscode\extensions\
ers\NR nayeem\Desktop\Dynamic algorithm.py" "
80

C:\Users\NR nayeem\Desktop>
```