M.Rohith
3122 21 5001 085

Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam - 603 110
(An Autonomous Institution, Affiliated to Anna University, Chennai)

UCS2403: DESIGN & ANALYSIS OF ALGORITHMS

Assignment 8

1. Given the adjacency matrix representation and the adjacency list representation of a simple graph $G = (V, E)$, write a Python program to perform:
   - Depth First Search (DFS) traversal on G
   - Breadth First Search (DFS) traversal on G
2. Given the adjacency matrix representation and the adjacency list representation of a simple graph $G = (V, E)$, write a Python program to perform:
   - Depth First Search (DFS) traversal on G
   - Breadth First Search (DFS) traversal on G
3. Given the adjacency matrix representation of a simple weighted, directed graph $G = (V, E)$, write a Python program to implement the Floyd-Warshall algorithm.
4. You are given a string S consisting of lowercase letters. Find the length of the longest palindromic subsequence in the string. A palindromic subsequence is a subsequence of the string that is read the same forward and backward. Implement a dynamic programming algorithm to solve this problem efficiently.
   Example: Input string: abacbca
   The solution is 5.

M.Rohith
3122 21 5001 085

1. **Program code for BFS:**

```
#Breadth-First Search (BFS) traversal
# of a graph in Python using both an adjacency list and an adjacency
matrix:

from collections import deque
import numpy as np

#-----------------------------------------------------------------
--------

# Using Adjacency List

# Create an adjacency list to represent the graph
graph = {
    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
}

# Define the BFS traversal function
def bfs_adj_list(graph, start):
    # Initialize visited, queue and result list
    visited = set()
    queue = deque([start])
    result = []

    # BFS algorithm
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)
            queue.extend(graph[node])

    return result

# Call the BFS traversal function and print the result
```

M.Rohith
3122 21 5001 085

```
print("Using adjacency list = ",bfs_adj_list(graph, 2))


#-------------------------------------------------------------------
-------------

# Create an adjacency matrix to represent the graph
graph = np.array([
    [0, 1, 1, 0],
    [0, 0, 1, 0],
    [1, 0, 0, 1],
    [0, 0, 0, 1]
])

# Define the BFS traversal function
def bfs_adj_matrix(graph, start):
    # Initialize visited, queue and result list
    visited = set()
    queue = deque([start])
    result = []

    # BFS algorithm
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)
            neighbors = np.where(graph[node] == 1)[0]
            queue.extend(neighbors)

    return result

# Call the BFS traversal function and print the result
print("Using adjacency matrix = ",bfs_adj_matrix(graph, 2))

#-------------------------------------------------------------------
----------

# Both implementations should produce the same result: [2, 0, 3, 1].
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8> python BFS.py
Using adjacency list =  [2, 0, 3, 1]
Using adjacency matrix =  [2, 0, 3, 1]
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8>
```

2. **Program code for DFS:**

```
# Depth-First Search (DFS) traversal
# of a graph in Python using both an adjacency list and an adjacency
matrix:

import numpy as np

#-------------------------------------------------------------------
--------

# Using Adjacency List

# Create an adjacency list to represent the graph
graph = {
    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
}

# Define the DFS traversal function
def dfs_adj_list(graph, start, visited=None, result=None):
    if visited is None:
        visited = set()
    if result is None:
        result = []

    # Base case
    if start not in visited:
        visited.add(start)
        result.append(start)
        for neighbor in graph[start]:
            dfs_adj_list(graph, neighbor, visited, result)
```

M.Rohith
3122 21 5001 085

```
    return result

# Call the DFS traversal function and print the result
print("Using adjacency list = ",dfs_adj_list(graph, 2))

#-----------------------------------------------------------------
--------

# Using Adjacency Matrix

# Create an adjacency matrix to represent the graph
graph = np.array([
    [0, 1, 1, 0],
    [0, 0, 1, 0],
    [1, 0, 0, 1],
    [0, 0, 0, 1]
])

# Define the DFS traversal function
def dfs_adj_matrix(graph, start, visited=None, result=None):
    if visited is None:
        visited = set()
    if result is None:
        result = []

    # Base case
    if start not in visited:
        visited.add(start)
        result.append(start)
        neighbors = np.where(graph[start] == 1)[0]
        for neighbor in neighbors:
            dfs_adj_matrix(graph, neighbor, visited, result)

    return result

# Call the DFS traversal function and print the result
print("Using adjacency matrix = ",dfs_adj_matrix(graph, 2))

#-----------------------------------------------------------------
--------
```

```
# Both implementations should produce the same result: [2, 0, 1, 3].

# Note that both implementations use recursion to traverse the
graph,
# which can be less efficient than using an iterative approach for
large graphs
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8> python DFS.py
Using adjacency list =  [2, 0, 1, 3]
Using adjacency matrix =  [2, 0, 1, 3]
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8>
```

3. **Program code for Floyd Warshall:**

```
# Python3 Program for Floyd Warshall Algorithm

# Number of vertices in the graph
V = 4

# Define infinity as the large
# enough value. This value will be
# used for vertices not connected to each other
INF = 99999

# Solves all pair shortest path
# via Floyd Warshall Algorithm


def floydWarshall(graph):
    """ dist[][] will be the output
       matrix that will finally
        have the shortest distances
        between every pair of vertices """
    """ initializing the solution matrix
    same as input graph matrix
    OR we can say that the initial
    values of shortest distances
    are based on shortest paths considering no
    intermediate vertices """
```

M.Rohith
3122 21 5001 085

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

    """ Add all vertices one by one
    to the set of intermediate
     vertices.
     ---> Before start of an iteration,
     we have shortest distances
     between all pairs of vertices
     such that the shortest
     distances consider only the
     vertices in the set
    {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a
      iteration, vertex no. k is
     added to the set of intermediate
     vertices and the
    set becomes {0, 1, 2, .. k}
    """
    for k in range(V):

        # pick all vertices as source one by one
        for i in range(V):

            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):

                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                dist[i][j] = min(dist[i][j],
                                 dist[i][k] + dist[k][j]
                                 )
    printSolution(dist)


# A utility function to print the solution
def printSolution(dist):
    print("Following matrix shows the shortest distances\
 between every pair of vertices")
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
```

import copy
dist=copy.copy(graph)

M.Rohith
3122 21 5001 085

```
                print("%7s" % ("INF"), end=" ")
            else:
                print("%7d\t" % (dist[i][j]), end=' ')
            if j == V-1:
                print()


# Driver's code
if __name__ == "__main__":
    """
                10
           (0)------->(3)
            |         /|\
          5 |          |
            |          | 1
           \|/         |
           (1)------->(2)
                3           """
    graph = [[0, 5, INF, 10],
             [INF, 0, 3, INF],
             [INF, INF, 0,   1],
             [INF, INF, INF, 0]
             ]
    # Function call
    floydWarshall(graph)
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8> python FloydWarshall.py
Following matrix shows the shortest distances between every pair of vertices
      0       5             8             9
    INF       0       3             4
    INF     INF       0       1
    INF     INF     INF       0
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8>
```

4. **Program code for Longest Palindromic Subsequence:**

```
# Dynamic Program approach

# A utility function to get max
# of two integers
def max(x, y):
```

M.Rohith
3122 21 5001 085

```
    if(x > y):
        return x
    return y

# Returns the length of the longest
# palindromic subsequence in seq
def lps(seq, i, j):

    # Base Case 1: If there is
    # only 1 character
    if (i == j):
        return 1

    # Base Case 2: If there are only 2
    # characters and both are same
    if (seq[i] == seq[j] and i + 1 == j):
        return 2

    # If the first and last characters match
    if (seq[i] == seq[j]):
        return lps(seq, i + 1, j - 1) + 2

    # If the first and last characters
    # do not match
    return max(lps(seq, i, j - 1),
               lps(seq, i + 1, j))

# Driver Code
if __name__ == '__main__':
    seq = "abacbca"
    n = len(seq)
    print("The length of the LPS is",ss
                   lps(seq, 0, n - 1))
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8> python Palindromic_subsequence.py
The length of the LPS is 5
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-8>
```