Report – Assignment 3 – COT5405 – Analysis of Algorithms

Submitted by:
Raghuveer Sharma Saripalli (UFID: 50946752)
Akshay Ganapathy (UFID: 36846922)

Question 1: Rod Cutting Problem

a. Psuedo code:

```
1  def rodCut(n, prices):
2      memArray = integer array of size n + 1
3      memArray[0] = 0
4      for (i from 1 upto n):
5          maxPrice = -infinity
6          for (j from 0 upto i - 1):
7              maxPrice = max(maxPrice, prices[j] + memArray[i-j-1]);
8          memArray[i] = maxPrice;
9
10     return memArray[n]
```

Description:

In this problem there are recurring sub-problems which are computed and stored in memory for the first time and reused when the same sub-problem occurs. This helps us to avoid computing the same sub-problem multiple times. The dynamic programming approach used for this problem is called bottom-up approach in which we compute from base case till *n*. Result is the one where we take max price among the iterations in every iteration.

1. Create memArray of size n+1 and set first element to 0.
2. Repeat the following for values from i = 1 till n:
    a. Set maxPrice to -infinity.
    b. For values from j = 0 to i-1 do the following:
        i. Calculate the maximum of max price and prices(j)+memArray(i-j-1)
    c. Set memArray(i) = maxPrice
3. The result will be in memArray(n).

b. Time and Space Complexity:

Time Complexity:

From 4th and 6th line of algorithm we get the following equation:

$$T(n) = 1 + 2 + 3 + \cdots + (n-2) + (n-1)$$

$$T\ (n) = \frac{(n-1)*n}{2}\ (Sum\ of\ n\ natural\ numbers)$$
$$T(n) = \frac{(n^2 - n)}{2}$$
$$T(n) = O\ (n^2)$$

Space Complexity:

From 2nd line of algorithm, we have created an array of n + 1 elements and rest of them a few variables.

$$S(n) = O(n)$$

Question 2: Dynamic-programming reflection

The similarity between Dynamic Programming and the Divide and Conquer approach are that both concepts break the problem into sub-problems. Furthermore, all the decisions of the sub-problems are combined to answer the original problem.

Divide and Conquer is used when the sub problems are independent of each other and the result of one sub-problem cannot be re-used elsewhere. Dynamic Programming is used when there might be overlapping sub-problems, i.e., the result of each sub-problem is stored so that it can be re-used whenever necessary. Divide and Conquer follows a top-down approach whereas Dynamic Programming follows a bottom-up approach.

Question 3: Shortest Path Counting

a. Psuedo code:

```
def shortestPath(N):
    dp = matrix of size (N x N)
    for i = 1 up to N: // 1 for i = 1 or j = 1
        dp[i][1] = 1
        dp[1][i] = 1
    for i = 2 up to N:
        for j = 2 up to N:
            dp[i][j] = dp[i][j - 1] + dp[i - 1][j]

    return dp[N][N]
```

Description:

We may presume that the rook starts out in the lower left corner of a chessboard with rows and columns numbered from 1 to 8 without losing generality. Let Q (i, j) be the number of shortest paths taken by the rook in the ith row and jth column from square (1,1) to square (i, j), where 1 <= i, j <= 8. Any such route would be made up of vertical and horizontal movements all aimed at the-same-target.
For any 1 <= i, j <= 8, Q (i, 1) = Q (1, j) = 1 is self-evident. In general, the shortest path to square (i, j) is either through square (i, j -1) or through square (i-1, j).

Hence the recurrence relation is as follows:

$$Q\ (i,j) = \begin{cases} Q(i,j-1) + Q(i-1,j), & 1 < i,j \leq 8 \\ 1, & i = 1\ or\ j = 1 \end{cases}$$

1. Create an empty NxN matrix.
2. Initialize all cells in first row and first column with value 1.
3. Looping for all values of i from 2 till N:
    a. Looping for all values from j = 2 to N:
        i. matrix (i, j) = matrix (i, j-1) + matrix (i - 1, j)
4. Result will be in matrix (N, N).

b. Time and Space Complexity:

Time Complexity:

From 6th and 7th line of algorithm we get the following equation:

$$T(n) = (n-1) + (n-1) + \cdots\ n-1\ times$$
$$T(n) = n^2 - 2n - 1$$
$$T(n) = O\ (n^2)$$

Space Complexity:

In line 2, an array of (N x N) elements are created. Therefore, the space complexity,

$$S(n) = O(n^2)$$

c. Using elementary combinatorics:

Any shortest path for a rook to move from one corner of a chessboard to the diagonally opposite corner can be thought of as 14 consecutive moves to adjacent squares, 7 of which being up while the other 7 being to the right. Hence, the total number of distinct shortest paths is equal to the number of different ways to choose 7 positions among the total of 14 possible positions, which is equal to,

$$ {}^{14}_{7}C = \frac{14\,!}{(14-7)\,!\;\;7\,!} $$

$$ {}^{14}_{7}C = \frac{14\,!}{7\,!\;\;7\,!} $$

$$ {}^{14}_{7}C = \frac{14 \text{ x } 13 \text{ x } 12 \text{ x } 11 \text{ x } 10 \text{ x } 9 \text{ x } 8 \text{ x } 7\,!}{7\,!\;\;7\,!} $$

$$ {}^{14}_{7}C = \frac{17297280}{5040} $$

= 3432

Hence, by combinatorics, the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner is 3432.

Question 4: Implementing dynamic programming algorithms:

a. Psuedo code

```
def solve(matrix):
    M = number of rows in matrix
    N = number of columns in matrix
    result = 0
    memory = matrix of size (M x N)
    rowIndex = 0, colIndex = 0
    for (i from 0 to M):
        memory[i][0] = (matrix[i][0] == 0)
        if (result < memory[i][0]):
            result = memory[i][0], rowIndex = i, colIndex = 0

    for (i from 0 to N):
        memory[0][i] = (matrix[0][i] == 0)
        result = max(res, memory[0][i])
        if (result < memory[0][i]):
            result = memory[0][i], rowIndex = 0, colIndex = i

    for (i from 1 to M):
        for (j from 1 upto N):
            if (matrix[i][j] == 0):
                memory[i][j] = min(memory[i][j-1],
                memory[i-1][j], memory[i-1][j-1]) + 1
            else:
                memory[i][j] = 0
            if (result < memory[i][j]):
                result = memory[i][j], rowIndex = i, colIndex = j
    return result, rowIndex, colIndex
```

```
28 def printMatrix(result, rowIndex, colIndex):
29     print ("Size of Maximum size square sub-matrix is: " + result)
30     if (result > 0):
31         print ("and the sub-matrix is: ")
32         for (i = rowIndex; i > rowIndex - result; i--):
33             for (j = colIndex; j > colIndex - result; j--):
34                 print(matrix[i][j] + " ")
35             print(newline)
```

Description:

In this bottom-up dynamic programming solution:

- First fill the first column in the matrix *memory*, of size is M x N, by inserting the value of '1' in the same location where a '0' is found in *matrix;* reason being '0' in matrix a 1x1 square and can be our possible answer. The variables *result, rowIndex, colIndex* are also updated simultaneously. The same step is repeated for the first row in *matrix.*
- For the subsequent rows, if a '0' is found in *matrix*, the min(memory[i][j-1], memory[i-1][j], memory[i-1][j-1])+1 is calculated and inserted in memory[i][j].
- Otherwise, memory[i][j] is filled with a '0' and subsequently, the result, rowIndex and colIndex variables are updated.
- At the end return the result, row, and column index variables. There is a separate function which prints the matrix using these returned values.

b. Running time:

In line 7 and 12 there are total M and N iterations, respectively. In line 18 and 19 there are M-1 * N -1 iterations. Therefore, running time is as follows:

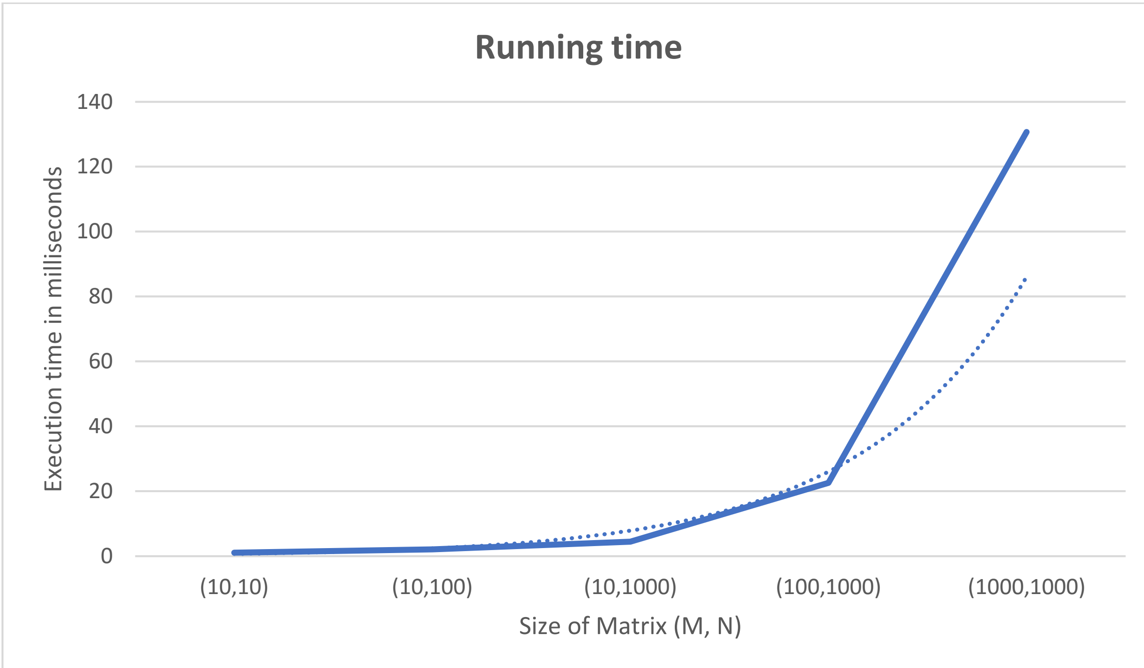
$$T(M, N) = M + N + (M - 1) \times (N - 1)$$

$$\approx M + N + (MN - M - N + 1)$$

$$\approx MN - 1$$

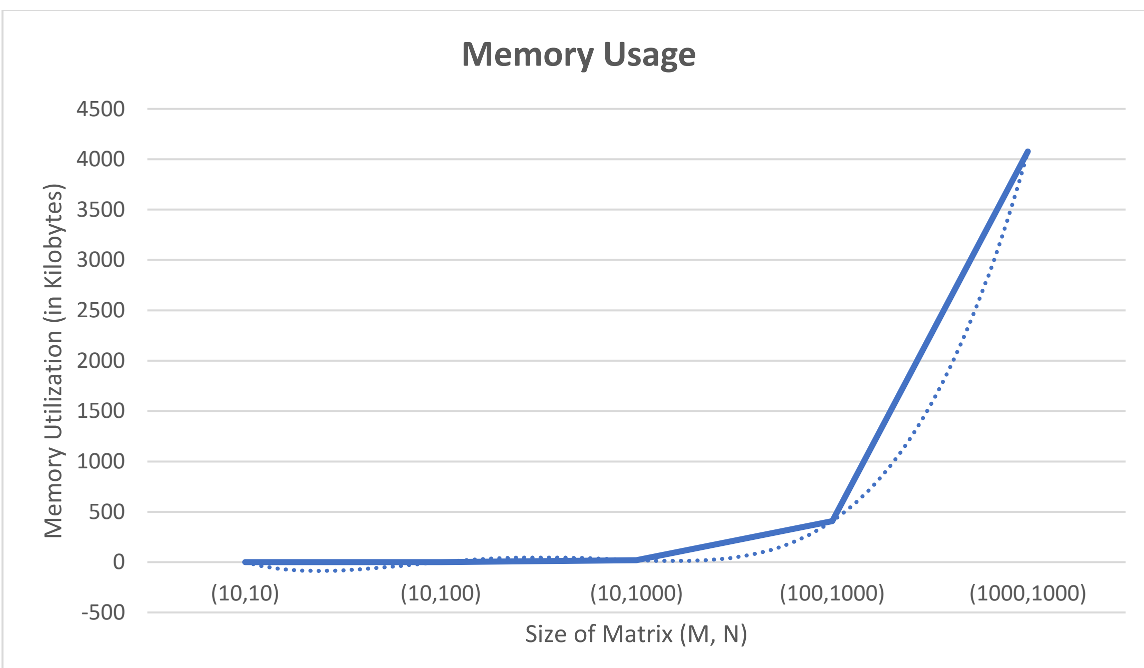
$$\approx O(MN)$$

c. Graph - Running time:



Above graph is plotted between Size of Matrix from the provided list [(10, 10), (10, 100), (10, 1000), (100,1000) and (1000, 1000)] and Execution time. Solid plot in the graph is actual execution time taken by the program and dotted plot is O (MN) general plot. Therefore, this proves the running time of the algorithm.

Graph – Memory Usage:



Above graph is plotted between Size of Matrix from the provided list [(10, 10), (10, 100), (10, 1000), (100,1000) and (1000, 1000)] and Memory Usage.

Solid plot in the graph is actual memory used by the program and dotted plot is O (MN) general memory utilization plot. Therefore, this proves the memory usage of the algorithm.