Assignment 4

Vaibhav Agarwal(B15139)

Q1. Assuming all the capacities are integers,
s denotes the source, **t** denotes the sink in network flow problem.
m denotes number of edges in Network Flow Graph
C = $\Sigma_{e \text{ out of } s}$ c_e. (Maximum possible flow out of the source).
The flow maintained by the Ford-Fulkerson Algorithm increases in each iteration,it increases by at least 1 in each iteration. Since it starts with the value 0, and cannot go higher than C, the While loop in the Ford-Fulkerson Algorithm can run for at most C iterations. Also BFS for finding a new path takes O(n+m) => O(m) time, so the time complexity of Max Flow algorithm will be **O(mC)**.

Psuedocode:

```
augment(f,P):
    (f , P)
    Let b = bottleneck(P, f)
    For each edge (u, v) ∈ P
        If e = (u, v) is a forward edge
            then increase f(e) in G by b
        Else ((u, v) is a backward edge, and
            let e = (v, u))
            decrease f(e) in G by b
        Endif
    Endfor
    Return(f)
Max-Flow:
    Initially f(e) = 0 for all e in G
    While there is an s-t path in the residual graph $G_f$
        Let P be a simple s-t path in $G_f$
        f' = augment(f , P)
        Update f to be f'
        Update the residual graph $G_f$ to be $G_{f'}$
    Endwhile
    Return f
```

Q2. We build up a table of solutions M, and we compute each of the values M[i, w] in O(1) time using the previous values. Thus the running time is proportional to the number of entries in the table. The algorithm correctly computes the optimal value of the problem, and runs in **O(nW)** time, the size of the solution matrix. There are two for loops outer which runs for n iterations and the inner loop which checks for every possible value of the weight limit w < = W, which runs W times.

Psuedocode:

```
Knapsack-Problem(n, W):
    Array M[0 . . . n,0... W]
    Initialize M[0, w]= 0  for each w = 0, 1, . . . , W
    For i = 1, 2, . . . , n
            For w = 0, . . . , W
                If $w < w_i$ then
                    M[i, w] = M[i − 1, w].
                Else
                    $M[i, w] = max(M[i − 1, w], v_i + M[i − 1, w − w_i])$.
                Endif
            Endfor
    Endfor
    Return M[n, W]
```

Q3.
The time complexity of above solution is **O(mn)** and auxiliary space used by the program is **O(mn)**. We used bottom up approach in this problem. We initialized the solution value of the smaller sub-problems and built the solutions of bigger problems using the subproblems.

Psuedocode:

```
LCS(X , Y):
    m <- length[X]
    n <- length[Y]

    for i = 1 to m
        c[i,0] = 0
    endfor

    for j = 1 to n
        c[0,j] <- 0
    endfor

    for i = 1 to m
        for j = 1 to n
            if (X[i] == Y[j]) then
                c[i,j] = c[i-1,j-1] + 1.
            else
                c[i,j] = max( c[i-1,j] , c[i,j-1] ).
            endfor
    endfor
    Return c[m][n].
```

Q4. There are O(n^2) pairs (i, j) for which this computation is needed; and for each pair (i, j), we can use the formula given at the beginning of this section to compute ei,j in O(n) time. Thus the total running time to compute all $e_{i,j}$ values is O(n^3). Following this, the algorithm has n iterations, for values j = 1, . . . , n. For each value of j, we have to determine the minimum in the recurrence to fill in the array entry M[j]; this takes time O(n) for each j, for a total of O(n^2). Thus the running time is **O(n^2)** once all the $e_{i,j}$ values have been determined.

<u>Slope</u>

$$= \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2}$$

<u>Intercept</u>

$$= \frac{\sum_i y_i - a \sum_i x_i}{n}$$

<u>Psuedocode:</u>

```
***Segmented-Least-Squares(n):***
    Array M[0 . . . n]
    Set M[0]= 0
    For all pairs i ≤ j
        Compute the least squares error ei,j for the segment pi,..., pj
    Endfor
    For j = 1, 2, . . . , n
        C_Min = MAX_VAL;
        For i = 1, 2, . . . , j
            C_Min = min (C_Min , $e_{i,j}$ + C + M[i − 1] ),
        M[j] = C_Min
    Endfor
    Return M[n]

***Find-Segments(j):***
    If j = 0 then
        Output nothing
    Else
        Find an i that minimizes $e_{i,j}$ + C + M[i − 1]
        Output the segment {pi,..., pj}
        Output the result of Find-Segments(i − 1) Endif
```

Q5.

Assuming both matrix are square matrix of same size N.
Addition and Subtraction of two matrices takes O(N^2) time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is **O($N^{\log 7}$)** (approximately O($N^{2.8}$))

<u>Psuedocode:</u>

```
Strassen(A,B):
        A = a b        //Break matrix A into 4 sub-matrix
            c d
        B = e f        //Break matrix B into 4 sub-matrix
            g h
        p1 = Strassen(a , f-h ) , p2 = Strassen(a+b , h ) ,
        p3 = Strassen( c+d , e) , p4 = Strassen(d , g-e)
        p5 = Strassen( a+d , e+h ) , p6 = Strassen( b-d , g+h ) ,
        p7 = Strassen(a-c , e+f ).

        C = p5+p6+p4-p2        p1+p2
            p3+p4              p1+p5-p3-p7
        return C.
```