Student Name: Yinsheng Dong
Student Number: 11148648
NSID: yid164
CMPT360 Assignment 3

1. Kleinberg and Tardos p. 317 #6

   The Good Printing Question.

   Solution:
   Assume that we have W words for good printing, and L for the maximum line length. The question has given the space after each word except the last, and c as the characters for each word, and the slack is the number of spaces left at the right margin.

   Firstly, we need to define the slack S. The slack function is for calculate how much space left when a word added.

```
// Create by yid164
// Slack function for Q1
// This function is for counting how many spaces left

function Slack(char_start, char_end):

    // if char_end - char_start is larger than max length, it return max
    if (char_end - char_start) > L:
        then return max

    spaceLeft = L - (char_end - char_start)

    return spaceLeft

end function.
```

   Then, we define an optimum solution to retrieve all the words and to find the minimum one. We start at the bottom line because the last word will contain the space value.

```
// Create by yid164
// The Opt function to retrieve all the words
// This function will return the sum of the squares of the slack of all lines

Function Opt[n]

    // if n is from start, then it is empty
    If n = 0:
        return 0
    Else
        // this for-loop is getting all the element from words
        For i = 1; i <= n; i++
            // this for-loop is for comparing every words and getting the minium
            // of Slack square + last Opt
            // This is the dynamic programming part
            For every j >= 0 and j <= i
                Opt[i] = min ((Slack(j, i))^2 + Opt[j-1])
            End for
        End for
    End if
    Return Opt[n]
End Function
```

The actual design algorithm from all above idea is:

```
// Create by yid164
// The Algorithm

// paragraph that given
vector<vector<char>> paragraph

// the maximum of length for each line
Length L

// line storing the character of the words
vector<char> lines

// Initially the length is 0
length = 0;

// this for loop is using to add each word to the line
for i = 1, i <= n, i++

    lines.add(Wi)

    if i < n

        length = length + Ci + 1
    else

        length = length + Ci
    end if
end for

// this is for pushing the line to the paragraph
while length <= L

        paragraph.add(lines)

// return the number of the line
return lineNum = paragraph.size()
```

In this solution, the Slack() function's time complexity is O(n), but the Opt() function's complexity is O(n^2), so the total time complexity is O(n^2)

2. Kleinberg and Tardos p. 323 # 11

The manufactures problem

Solution:

Company A charges a fixed rate r per pound: A = rs_i
Company B charges every week: B = c

Let Opt(i) be the cost for the optimal schedule from weeks 1 to i

Company A: Opt(i) = r * s_i + Opt(i-1)
Company B: Opt(i) = 4 * c + Opt(i-4)

Then we need to determine which one is smaller when we choose combine company A and company B. However, if we choose company B and the i is smaller than 4, we won't get the maximum benefit. In this case, we have to restrict the i is smaller or equal to i.

Since Company B iterates the pointer after 4 point, so we have to know the case i = 0, 1, 2, 3, and 4, then we can use the Opt(i) = min ((r * s + Opt(i-1)), (4 * c + Opt(i-4)).

Then, the algorithm is:

```
// Create by yid164
// The Algorithm Design for Q2
// The Optimal function for Q2

// The Supply list that company have
S[0...n]

// The best choice to return
BestChoice[]

// Opt function
Opt(n):

    // if there is nothing in the supply weeks
    if n == 0:
        Opt(n) = 0
        BestChoice[].add(Opt(n))

    // if the supply chain has 4 or less weeks then we have to decide which company is better
    // we do that because when the weeks are less than 4, choosing company B might not behave
    // the best solution
    Elif n <= 4:
        for i = n, i>=0; i--:
            Opt(n) = min ((S[n] * r + Opt(n-1)), 4c)
            BestChoice[].add(Opt(n))
        endFor

    // if n is larger than 4, then do the DP
    // Iterate all element that larger than 4, and
    // Add the best choice of Opt(i)
    Elif n > 4:
        for i = n, i >= 0; i++:
            Opt(n) = min((S[n] * r + Opt(n-1)), 4c+Opt(n-1))
            BestChoice[].add(Opt(n))
        endFor
    EndIf

    // return the best choice
    return BestChoice[]
```

In this algorithm, the first if statement contains O(1) complexity, and the second if statement contains O(n^2) complexity, and the finally if statement is O(n^2) complexity. Then the total time complexity is O(n^2).

3. The matrix problem

   We assume that there is a new matrix N which is the largest balanced quite submatrix in M. The size of matrix N initially is same as the matrix M (which means for all M [i] [j] = 'q'). Then we need to check every item in M to know which one is 'q', and the best way to start is from the right bottom corner. Also, we need to fit the N[i][j] which 1<= i <= n and 1<= j <= m, which n is the rows of M and m is the columns of M.

   For convenience in this question, we assume that the quite = 1, and the noisy = 0.

To do the dynamic programming algorithm, there are some variables declared below:

```
// Create by yid164
// The Algorithm variables of Q3

// Q is the size
Q = 0;

// M is the matrix that question given
M = the matrix that question given

// the row of M
n = the rows of M

// the column of M
m = the col of M

// the submatrix
N = the submatrix of M

// left pointer in current point
LP = the current point - 1

// upper pointer in current point
UP = the current point - 1
```

These variables are defined for the algorithm

Next, we are to make an algorithm to set up the N by using fixed rows

```
// Create by yid164
// This algorithm is for determine the column variables in fixed row
i = n

while i >= 1:

    // from the 1st to the final row, if the M[i][j] is noisy
    // then the N in the same position and it's upper position will be noisy
    if M[i][0] == 0:
        N[i][0] = 0
        N[i][UP] = 0

    // If the M[i][j] is quite, then set up the N's same position and its upper point
    // be quite
    else:
        N[i][0] = 1
        N[i][UP] = 1
    endif

    i--
end while
```

This is the base case for fixed rows.

Next, an algorithm should define the base case for fixed column:

```
// Create by yid164
// This algorithm is for determine the row variables in fixed column
j = m

while j >= 1:

    // from the 1st to the final column, fi the M[i[[j] is noisy
    // the the N in the same position and it's left position will be noisy
    if M[0][j] == 0:
        N[0][j] = 0
        N[j][LP] = 0

    // Else, set up the N's same position and its left point be quite
    else:
        N[0][j] = 1
        N[j][LP] = 1
    endif

    j--
end while
```

Then, we will have the DP algorithm:

```
// Create by yid164
// This algorithm is for comparing the variables
s = n
k = m

while s >= 2:
    while k >= 2:

        // If the M[s][k] = noisy, then do the DP then
        // N in same position and upper row and left col = the minimal one
        // so we can fit 1 <= s <=n and 1<= k <= m
        if M[s][k] = 1
            N[s][k] = min ( N[s-1][k-1], N[s-1][UP], N[k-1][LP] ) + 1
            N[i][UP] = N[s-1][UP] + 1
            N[j][LP] = N[j-1][LP] + 1
        else:
            N[s][k] = 0
            N[s][up] = 0
            N[s][left] = 0
        endif

        // let Q equal to N and return Q
        if( Q < N[s][k] )
            Q = N[s][k]
        endif
        k --
    end while
    s --
end while

// the Q is the final answer of the submatrix
return Q
```

The final algorithm is the DP, for comparing each one which s, k >=2 to make sure matrix N is balanced and maximum.

The time complexity of Algorithm 1 is O(n), the Algorithm 2 is O(n), and the DP is O(n^2). Then the totally time complexity is O(n^2).