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.
First, we need to set the boundary condition when the M[i][j] size is 1, which means the matrix M only have 1 item, 1 row or 1 col. In next page

```
// Create By yid164
// Matrix M[i][j] shows the right bottom item of the M
// Q is the submatrix size
// if one of i, j is 0, or both are 0
// which means it does have only 1 row or 1 column
// or it does have only 1 element

// Q is the submatrix size
Q = 0

//if M only has 1 row, if there an element is quite, then Q = 1
if i == 0 && j > 0:
    while k <= j:
        if M[i][k] == 1:
            Q = 1
            return
        else:
            Q = 0
        end if
        k++
    end while

//if M only has 1 col, if there an element is quite, then Q = 1
else if j == 0 && i > 0:
    while k <= i
        if M[k][j] == 1:
            Q = 1
            return
        else:
            Q = 0
        endif
        k ++
    end while

//if M only has 1 element, if that element is quite, then Q = 1
else if i == 0 && j == 0:
    if M[i][j] == 1:
        Q = 1
    else:
        Q = 0
    end if
endif
```

Then, we can design the algorithm:

```
// create by yid164
// Algorithm for DP

// M[i][j] is the right bottom corner element
// N[i][j] is an assumed matrix that store the submatrix of M

// the maximum size of balanced quite matrix
Q = 0

// pointer of row
a = 1

// pointer of col
b = 1

// do the recursive DP for adding the quite element from M to N
// n is the matrix M's max row number
// m is the matrix M's max col number
while a <= n
    while b <= m

        // if the M[a][b] is quite, then add the min (left, up, and tranversal item) + 1
        // if all the items is quite, then the N[a][b] becomes to 2
        // if one of the items is noisy, then the N[a][b] reminds to 1
        if M[a][b] == 1:
            N[a][b] = min (M[a-1][b], M[a][b-1], M[a-1][b-1]) + 1

        // if the M[a][b] is noisy, then N[a][b] is noisy (0)
        else:
            N[a][b] = 0;
        endif

        // add N[a][b] to Q
        if( Q < N[a][b] )
            Q = N[a][b]
        endif
        b ++
    end while
    a ++
end while
return Q
```

This algorithm retrieves all the element from M(1,1) to the last item, and to adjust all the items recursively, then get the maximum of the quite balanced submatrix of M.

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