Atanas Dimitrov
Homework #3
CSCI 6470
Dr. Cai

Solution to problem 1:

A recursive solution of the assembly line problem gives us overlapping problems. In the case of such solution which was briefly mentioned in Section 15-1 of our textbook. Coming up with the answer will involve re-computation of the same sub problems on multiple occasions which makes to Algorithm run in 2^(n-j) time complexity. The cause of this is that at each step in the recursion illustrated in the textbook we will have to recalculate some of the optimal subproblems again and again. For example consider:

f1[j]=e1+a1,1 if j=1
f1[j]= .....

and the recursions for f2[j]. At some point in time the minimal path may be the same but we will be calculating them twice once for f1[j] and once for f2[j] as i and j decrease since recursive calls cannot communicate. This problem resembles the overlapping problems in Exercise 3 of this homework if recursion was used.

Solution to problem 2:

1. Solution characterization and optimal substructure.

For the number of extra spaces we have M - j + i - (k=i to j)∑l_k. Thus a valid solution will consist of an arrangement of the words so that the upper is ≧ 0 and this holds for each line in the output paragraph. We can use one-dimensional array s[1..m] in which each element say s[x] (1<=x<=m m being the last line number) will contain the number of the last word on row x relative to the beginning of the list l1. Now to redefine the requirement M - j + i - (k=i to j)∑l_k in terms of the solution we express this entity using s[] thus defining a plausible solution. We can replace j with s[x] since they both point to the same element by our setup. "i" we can replace with say s[y] for the same reason, where the range of y and x is the same only x>y. We notice that j-i gives us the number of spaces required between the words on the line. Let's ignore the sum for now. Thus, the number of extra spaces can be rewritten as:

M - (s[x] -s[y]) + (k=i to j)∑l_k)

We notice that s[y] is nothing but s[x-1]+1 since s[x-1] points to the last word on the previous line adding 1 will take us to our current row - first word. Thus our equation is now:

M - (s[x] -(s[x-1] + 1)) + (k=i to j)∑l_k)

Looking at the sum we know we can replace i with s[x-1] + 1, we also know how to replace j, then the above can be rewritten as:

M - (s[x] -(s[x-1] + 1)) + (k= (s[x-1] + 1) to s[x])∑l_k)

Then in order for a solution to be correct then we must ensure that:

M - (s[x] -(s[x-1] + 1)) + (k= (s[x-1] + 1) to s[x])∑l_k) ≧ 0, which will be always true as long as M - (s[x] -s[y]) + (k=i to j)∑l_k) ≧ 0 since we have used substitution.

The quantity we want to minimize for each line is:

(x=1 to m-1)∑M - (s[x] -(s[x-1] + 1)) + (k= (s[x-1] + 1) to s[x])∑l_k)

In our setup the optimal substructure does exist. In other words if we have obtained the minimum of the empty spaces at the end of each line up to line x then we have arrived there by obtaining the

same minimum up to line x-1. Let us suppose that we have arrived at s[x] (x<m) and we have made choices that minimize the sum of empty spaces left over on each row to absolute minimum, by determining this same minimized quantity for s[x-1] and row x. Suppose that there is another way to minimize the sum over lines 1 to x-1. Then we could substitute this arrangement in the minimized solution for x to make it even more minimized, which contradicts our assumption that the latter is already at the absolute minimum.

2. Recursive formula.

Let F(j) calculate the optimal arrangement of words 1 to j. Then if we let i be the first word on the row which ends with word j our solution can be expressed as $F(i-1) + M - j + i - (k=i \text{ to } j)\sum l_k$. We cannot specifically determine the value for i right away but we can certainly state that i will be the smallest value between 1 and j such that $M - j + i - (k=i \text{ to } j)\sum l_k >= 0$. Suppose that this value is p. Then suppose we have function price(i,j) which finds the minimal cost for words i to j to included on the same row. Then price(i,j)=0 if we have reached the last word namely j=n. If the end of words is not reached the value will be $M - j + i - (k=i \text{ to } j)\sum l_k$ in Thus the recursive formula can be expressed as:

F(j)=0 when j=n
F(j)=(p<=i<=j) min (F(i-1) + price(i,j)) j!=n

3. Algorithm.

This algorithm simply take advantage of the dynamic programming approach to accomplish the above recurrence. We calculate the first line first and keep the rest a s a subproblem. Since all the sums are minimized on the lines we have applied the algorithm then the total sum is also minimized.

```
NEAT(n, l[], M, s[])
        i←1
        curr_line←1

        while i!=n
                j←i
                line_sum←0
                while M - line_sum ≧0 and j<n
                        line_sum←line_sum+l[j] + j - i
                        j ← j+1
                s[curr_line]←j-1
                curr_line←curr_line+1
                i←j
        s[curr_line]←i
        return s, curr_line
```

4. Constructing the solution.

The solution is contained in s as previously specified. We can use a function to print the text. curr_line is the last line in the paragraph.

```
PRINT (n, l[], s, curr_line)
```

```
i←0
for line←1 to curr_line
        while i < s[line]-1
                print l[i] " "
                i++
        print NEW_LINE_CHARACTER
        line++
```

5. Time analysis.
NEAT has two loops in its body. Thus intuitively we can say the it is O(n^2). Although there is a tighter bound on its time complexity. The running time is proportional to the input due to the fact that the function goes over each element only once. The rest of the operations will give us some constant time proportional to the number of elements such that we have c*n operations performed. Thus we can surely state that the time complexity is O(n). For PRINT we have the same situation. Although we have two nested loops the algorithm will go through each word once. Thus its time complexity is O(n) as well. Summing these up we get T(n)= O(n)+O(n) = O(n).

Solution to problem 3:

1. Solution characterization and optimal substructure.
There are n rows and n columns. Say r is the r-th row and c is the c-th column where 1<=c,r<=n. We can say that this problem can be divided into similar sub-problems. Each sub-problem we can define as finding the most rewarding way to get to square [r,c]. We clearly have the optimal substructure. Suppose we have obtained the most rewarding way to get to [r,c], going back a row we could have taken three different steps each of which ends up at [r,c]. They will all be on the previous row and differ in the column number, namely: [r-1, c-1], [r-1, c], [r-1, c+1]. Suppose we have found the most profitable path from row 1 to [r,c]. then suppose that there is a more profitable path through either [r-1, c-1], [r-1, c], [r-1, c+1], then we can substitute this path in our optimal path and get a more profitable path, statement which contradicts our initial assumption that the path to [r,c] is optimal. We know that there is no profit on the bottom row, i.e. p(1,c) is 0 for all c $\in$(1..n).

2. Recursive formula.
We can now define a recursive formula for the solution. We can reduce the problem by 1 row each iteration thus let F(r, c) be profit of the most profitable path to [r,c] then:

$F(1, c) = 0$ for all $1 \leqq c \leqq n$

$F(r,c) = 1.\ F(r-1, c+1) + p([r-1, c+1], [r,c])$
$2.\ F(r-1, c) + p([r-1, c], [r,c])$
$3.\ F(r-1, c-1) + p([r-1, c-1], [r,c])$

Case 1 can only hold given we are not at the last column i.e. we must have c<n. Case 3 can only hold given we are not at the first column i.e. we must have c>1.

3. Algorithm.
temp

```
>> we first set to 0 the bottom row
c←1
do
        max-profit-matrix[1, c]←0
while c≦n
for r ←2 to n
        for c ←1 to n
                temp←max-profit-matrix[r-1, c] + p((r-1, c), (r,c))
                if c > 1
                        if max-profit-matrix[r-1, c-1] + p((r-1, c-1), (r,c)) > temp
                                temp←max-profit-matrix[r-1, c-1] + p((r-1, c-1), (r,c))
                if c < n
                        if max-profit-matrix[r-1, c+1] + p((r-1, c+1), (r,c)) > temp
                                temp←max-profit-matrix[r-1, c+1] + p((r-1, c), (r,c))
                max-profit-matrix[r, c]←temp
```

This algorithm however doesn't produce the path only the max profit at the top of the matrix at some column c.
Thus we will need to design a way to keep track of the indexes as well. Suppose we have matrix index[r,c] then index[r,c] is the column number in the path 1 row down. Thus our solution becomes:

```
MAX-PROFIT(p(), n)
temp
>> we first set to 0 the bottom row
c←1
do
        max-profit-matrix[1, c]←0
while c≦n
for r ←2 to n
        for c ← 1 to n
                temp←max-profit-matrix[r-1, c] + p((r-1, c), (r,c))
                index[r,c]←c
                if c > 1
                        if max-profit-matrix[r-1, c-1] + p((r-1, c-1), (r,c)) > temp
                                temp←max-profit-matrix[r-1, c-1] + p((r-1, c-1), (r,c))
                                index[r,c]←c-1
                if c < n
                        if max-profit-matrix[r-1, c+1] + p((r-1, c+1), (r,c)) > temp
                                temp←max-profit-matrix[r-1, c+1] + p((r-1, c), (r,c))
                                index[r,c]←c+1
                max-profit-matrix[r, c]←temp
return max-profit-matrix index
```

4. Constructing the solution.
Now that we have the 2 matrices we must design an algorithm to collect the solutions. As I mentioned earlier the top row contains the max profit and thus is the largest number. Thus we can have the following setup:

```
MAX-PROFIT-PATH(index)
        r←n
        c← Sequential Search for max element on row n in index[][]
        PRINT (r, c)

PRINT (r, c)
        if r>1
                PRINT (r-1, index[r,c])
        display " " i "/" j " "
```

5. Time analysis.
We have 3 loops in MAX-PROFIT. The first (while) loop will take O(n).
Third (for) loop is a sub-loop of the second (for) loop thus they will take at max O(n^2) together.
Thus T(n) for MAX-PROFIT = O(n^2) + O(n) = O(n^2)
For MAX-PROFIT-PATH: the sequential search will take O(n), and PRINT will print 1 element from each row thus O(n). So deriving the solution will take O(n) + O(n) = O(n)
All combined the algorithms will take T(n) = O(n^2) + O(n) = O(n^2)