CSCI-GA.1170-001/002 Fundamental Algorithms November 14, 2019

Solutions to Problem 1 of Homework 8 (16 (+6) Points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, November 14*

Collaborators: NetID1, NetID2

You may or may not be old enough to remember how people used to write text messages a while back: on a standard issue phone, the $N = 26$ letters would be assigned in groups of three or four to the $K = 8$ keys ② to ⑨ on the dial pad, and in order to type the l^{th} letter on a particular key, one would tap that key l times in rapid succession.

Consider a text for which the frequency $f_i \in \mathbb{N}$ of each letter $i = 1, \ldots, N$ is known.[1] Typing this text on a phone would require $T = \sum_{i=1}^{N} p_i \cdot f_i$ taps, where p_i is the position of letter i on its key. Clearly, if one wishes to minimize T, the optimal assignment of letters to keys—in alphabetical order—depends on the text in question; more precisely, on the frequencies f_i. In particular, having groups of just three or four letters may be suboptimal.

In this task you will develop an algorithm that given (general) K, N, and frequencies $f_1, \ldots, f_N$ finds an assignment of the letters $1, \ldots, N$ in alphabetical order to the keys $1, \ldots, K$ such that T is minimized.

(a) (4 points) For $1 \le i \le j \le N$, let $C[i, j] := \sum_{k=i}^{j} (k - i + 1) \cdot f_k$, which is the number of taps required to type only the letters $i, \ldots, j$ in the text if they are all assigned to a single key (in alphabetical order).

Provide pseudocode of an $O(N^2)$ algorithm that fills the entire array $C[\cdot, \cdot]$. Ignore cells $C[i, j]$ where $i > j$.

Solution:

```
1 FillArrayC(N, C, f)
2     for i= 1 to N
3         for j = i to N
4             C[i,j] = (j − i + 1) · f[j] + C[i, j − 1]
5     return C
```

This is a $O(n^2)$ runtime code. □

For $1 \le n \le N$ and $1 \le k \le K$, define $T[n, k]$ to be the smallest possible number of taps required to type only the letters $1, \ldots, n$ in the text if they are assigned to keys $1, \ldots, k$.

(b) (2 points) For $n = 1, \ldots, N$, what is $T[n, 1]$ (in terms of C)?

Solution:

$$T[n, 1] = C[1, n]$$

□

[1] For simplicity letters are denoted by numbers in the remainder of this task.

(c) (5 points) Derive a recurrence relation for $T[n,k]$ and justify it. Note that in (b) you considered the base case of this recurrence.

Solution:

$$T[n,k] = \begin{cases} C[1,n] & \text{k=1} \\ \min\limits_{k-1 \le i \le n-1} T[i,k-1] + c[i+1,n] & 2 \le k \le K \text{ and } k \le n \le N \end{cases} \quad (1)$$

□

In the following you may assume that you have direct access to the filled array $C[\cdot,\cdot]$.

(d) (4 points) Devise a polynomial-time (in N and K) algorithm PHONE-TD(N,K) (write pseudocode) that fills table $T[\cdot,\cdot]$ in a top-down, recursive fashion *with memoization*. Fill in the blanks to complete the algorithm.

```
1 PHONE-TD(N, K)
2     initialize T[·,·] with all entries ∞
3     return PHONE-TD-AUX(N, K, T)

1     PHONE-TD-AUX(N, K, T)
2         if T[N, K] ≠ ∞
3             .....
4         if .....
5             q = .....
6         else
7             q = .....
8             for l = ..... to .....
9                 q = .....
10        T[N, K] = q
11        return q
```

Solution:

```
1 PHONE-TD(N, K)
2     initialize T[·,·] with all entries ∞
3     return PHONE-TD-AUX(N, K, T)

1     PHONE-TD-AUX(N, K, T)
2         if T[N, K] ≠ ∞
3             return T[N, K]
4         if K = 1
5             q = C[1, N]
6         else
```

```
7               q = ∞
8               for l = K − 1 to N − 1
9                   q = min(q, Phone − TD − Aux(l, K − 1, T) + c[l + 1, N])
10          T[N, K] = q
11          return q
```

□

(e) (6 points) [**Extra credit**] Finally, develop a procedure PHONE-FIND(N, K) (write pseudocode) that uses the filled $T[\cdot, \cdot]$ to output an assignment of the letters to the keys such that the text can be typed with $T[N, K]$ taps. Write pseudocode and analyze the running time of your algorithm. (**Hint**: It might make sense to devise a Bottom Up Algorithm that constructs T first and also construct a helper matrix L. These matrices will be available to the algorithm PHONE-FIND. Note that you will only need to use L to construct the solution but you will need T to construct L in a bottom-up fashion.) .

Solution:

```
1   PHONE-FIND(N, K)
2   let T[N][K] and L[N][K] to be new array
3   Fullfill(T,L,N,K)
4   Print-allocation(L)
```

```
1   FULLFILL(T, L, N, K)
2       for i =1 to N
3           for j = 1 to K
4               if j=1
5                   T[i][j] = C[1,i] and L[i][j] = 0
6               else q = ∞
7                   for l = j − 1 to i − 1
8                       if q >T[l][j-1]+c[l+1][N]
9                           q = T[l][j − 1] + c[l + 1][N]
10                          L[i][j] = l + 1
```

```
1   PRINT-ALLOCAITON(L, N, K)
2       for i =K to 1
3           l = L[N][i]
4           Print(l)
5           N=l-1
```

In my code, the fullfill process have three loops, the first is from 1 to N, the second is from 1 to K, the third is from j-1 to i-1 □

CSCI-GA.1170-001/002 Fundamental Algorithms November 14, 2019

Solutions to Problem 2 of Homework 8 (12 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, November 14*

Collaborators: NetID1, NetID2

Some of us are familiar with the tale of a pot of gold at the end of every rainbow, guarded by a wily leprechaun. Imagine that you have somehow gone to the end of the rainbow and you are confronted with n pots labeled $[1 \dots n]$ which is guarded by a very smart leprechaun. Each pot has a value v_i attached to it. The leprechaun is annoyed that you have found the pot of gold but admires your bravery. The leprechaun challenges you to play a game: Both of you take turns picking a pot and you can leave with whatever you have chosen. The leprechaun graciously agrees to go second but there is a catch. You can only pick a pot either from the beginning or the end.

The leprechaun is as smart as you. When it is the turn of the leprechaun, it will choose its pot so as to *minimize* what you can get from the remaining choices. You will use Dynamic Programming to formalize a strategy to win the game. You may assume that n is even.

(a) (3 points) Consider the following "greedy strategy". You pick the best possible pot at each turn, i.e, you pick the pot with the higher value. Give an example where $n = 4$ where this strategy is not optimal. Argue why this strategy is incorrect.

Solution: When it is 1,2,6,4. Then the first round i get 4. Then 1,2,6 is left. The leprechaun get the 6. 1 and 2 are left. Then i get 2. The leprechaun get 1. Finally, i get 4 and 2. The leprechaun get 6 and 1, which is greater than me. Then this strategy is not optimal.

This algorithm is not optimal because it only consider the current situation without considering what will happen next. Actually, there may be some situations that firstly we take some small ones and then we can take some big ones. The sum of these two numbers can be bigger than the sum of both rounds taking the bigger one. □

(b) (5 points) Formulate a recursive equation and argue the correctness of its optimality. You will use a two dimensional array. Clearly, define the meaning of $M[i, j]$ to help formulate this relation. (**Hint**: Do not forget that the leprechaun makes its move in such a way that your winnings are minimized from the remaining choices.)

Solution: Define M[i,j] as the maximum value i can get from pot i to pot j. $x = pots[i] + min(M[i+2, j], M[i+1, j-1])$,$y = pots[j] + min(M[i+1, j-1], M[i, j-2])$

$$M[i,j] = \begin{cases} pots[i] & \text{i=j} \\ max(pots[i], pots[j]) & \text{j=i+1} \\ max(x, y) & 1 \le i \le n-2 \text{ and } i+2 \le j \le n \end{cases} \tag{2}$$

□

(c) (4 points) Write a bottom-up (iterative) algorithm to solve the problem and return the maximum value possible. This will implement the above recursive formulation. What is the runtime of your algorithm? It will take as input the array V of values and size n of the array V. Fill in the blanks in the template below to complete the bottom-up algorithm.

```
1 Initialize n × n matrix M.
2 UnderTheRainbow(V, n)
3     for i = 1 to n − 1 //Initialize the base cases here.
4         ..............
5         ..............
6     M[n][n] = ............
7     for i = n − 2 to 1
8         for j = ......to ..........
9             value1 = ..............
10            value2 = ..............
11            value3 = ..............
12            case1 = ..............
13            case2 = ..............
14            M[i][j] = max(case1, case2)
15    return M[1][n]
```

Solution:

```
1 Initialize n × n matrix M.
2 UnderTheRainbow(V, n)
3     for i = 1 to n − 1 //Initialize the base cases here.
4         M[i][i] = V[i]
5         M[i][i+1] = max(V[i],V[i+1])
6     M[n][n] = V[n]
7     for i = n − 2 to 1
8         for j = i + 2to n
9             value1 = M[i + 2][j]
10            value2 = M[i + 1][j − 1]
11            value3 = M[i][j − 2]
12            case1 = V[i] + min(value1, value2)
13            case2 = V[j] + min(value2, value3)
14            M[i][j] = max(case1, case2)
15    return M[1][n]
```

The runtime of my code is $O(n^2)$ □

CSCI-GA.1170-001/002 Fundamental Algorithms November 14, 2019

Solutions to Problem 3 of Homework 8 (13 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, November 14*

Collaborators: NetID1, NetID2

You are a CFO of a baby sitting company, and got a request to baby sit n children one day. You can hire several babysitters for a day for a fixed cost B per babysitter. Also, you can assign an arbitrary number of children $i \geq 1$ to a babysitter. However, each parent will only pay some amount $p[i]$ if his child is taken care of by a babysitter who looks after i children. For example, if $n = 7$ and you hire 2 babysitters who looks after 3 and 4 children, respectively, you revenue is $3p[3] + 4p[4] - 2B$.

Given $B, n, p[1], \ldots, p[n]$, your job is to assign children to babysitters as to maximize your total profit. Namely, you want to find an optimal number k and an optimal partition $n = n_1 + \ldots + n_k$ so as to maximize revenue $R = n_1 \cdot p[n_1] + \ldots + n_k \cdot p[n_k] - k \cdot B$.

(a) (5 points) Let $R[i]$ denote the optimum revenue you can get by looking after i children. E.g., $R[0] = 0$, $R[1] = p[1] - B$, $R[2] = \max(2p[1] - 2B, 2p[2] - B)$, etc. Write a recursive formula for $R[n]$ in terms of values $R[j]$ for $j < n$.

Solution:

$$R[n] = \begin{cases} 0 & \text{n=0} \\ p(1) - B & \text{n=1} \\ \max\limits_{1 \leq k \leq n-1} R[k] + R[n-k], np(n) - B & 2 \leq n \end{cases} \tag{3}$$

□

(b) (4 points) Write an iterative bottom-up procedure to compute the optimal revenue.

Solution:

```
2 optimalRevenue(B, n, p)
3     if n=0 then return 0
4     if n=1 then return p[1]-B
5     let R[n] to be new array,all of them equal to −∞
6     R[0] = 0,R[1] = p[1]-B
7     for i = 2 to n
8           for j = 1 to i-1
9                 R[i] = max(R[j]+R[n-j ],R[i])
10          R[i] = max(R[i],ip[i]-B)
11    return R[n]
```

□

(c) (4 points) Explain how to augment your procedure (in part (b)) to also compute the optimal number of babysitters k and the actual partition of children. Either English or pseudocode will work.

Solution: In this pseudocode the R[n][1] record the optimal revenue. R[n][2] record the optimal number of babysitters. R[n][3] record the optimal partition. And R[n][3] is also an array. Here we denote it as Count[n].If Count[i] = j means each of these j babysitters will take care of i babies.

```
2 OPTIMALREVENUE(B, n, p)
3     if n=0 then return 0
4     if n=1 then return p[1]-B
5     let R[n][3] to be new array,all of R[n][1] equal to −∞
6     R[0][1] = 0,R[0][2] = 0,R[1][1] = p[1]-B,R[1][2] = 1,R[1][3][1]=1
7     for i = 2 to n
8           for j = 1 to i-1
9                 if R[j][1]+R[n-j][1]>R[i][1]
10                      R[i][1] = R[j][1]+R[n-j][1],R[i][2] = R[j][2]+R[n-j][2],and all of R[i][3] set to zero
11                      for a =1 to n
12                            R[i][3][a] = R[j][3][a]+R[n-j][3][a]
13          if ip[i]-B >R[i][1]
14                R[i][1] = ip[i]-B,R[i][2] = 1,and all of R[i][3] set to zero
15                R[i][3][i]=1
16    return R[n]
```

□