CSCI-GA.1170-001/002 Fundamental Algorithms November 5, 2014

Solutions to Problem 1 of Homework 7 (6 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 5*

Using dynamic programming, find the optimum printing of the text *"Not all those who wander are lost"*, i.e. $\ell_1 = 3, \ell_2 = 3, \ell_3 = 5, \ell_4 = 3, \ell_5 = 6, \ell_6 = 3, \ell_7 = 4$, with line length $L = 14$ and penalty function $P(x) = x^3$. Will the optimal printing you get be consistent with the strategy "print the word on as long as it fits, and otherwise start a new line"? Once again, you have to actually find the alignment, as opposed to only finding its penalty.

Solution:
The optimal solution to this problem is the Align the words this way.

```
Line 1:"Not _ all _ those _"            // Ending spaces = 1
Line 2:"who _ wander _ _ _ _ "          // Ending spaces = 4
Line 3:"are _ lost _ _ _ _ _ _"         // Ending spaces = 6
```

The ending paces on Line 1 is 1, Line 2 is 4, and Line 3 is 6.
So the total penalty is $(1)^3 + (4)^3 + (6)^3 = 281$

This shows that the greedy option of "if it fits, it prints" is not the optimal option. This is because the last line will have a massive amount of ending spaces, giving a large penalty. Here is this shown.

```
Line 1:"Not _ all _ those _"         // Ending spaces = 1
Line 2:"who _ wander _ are "         // Ending spaces = 0
Line 3:"lost _ _ _ _ _ _ _ _ _ _"    // Ending spaces = 10
```

So the total penalty is $(1)^3 + (0)^3 + (10)^3 = 1001$

The greedy approach gives a Penalty of 1001 where as the optimal give the penalty of 281, so no the greedy approach is not optimal.

□

CSCI-GA.1170-001/002 Fundamental Algorithms November 5, 2014

Solutions to Problem 2 of Homework 7 (20 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 5*

Let $X[1 \dots m]$ and $Y[1 \dots n]$ be two given arrays. A common supersequence of X and Y is an array $Z[1 \dots k]$ such that X and Y are both subsequences of $Z[1 \dots k]$. Your goal is to find the *shortest* common super-sequence (SCS) Z of X and Y, solving the following sub-problems.

(a) (4 points) First, concentrate on finding only the length k of Z. Proceeding similarly to the longest common subsequence problem, define the appropriate array $M[0 \dots m, 0 \dots n]$ (in English), and the write the key recurrence equation to recursively compute the values $M[i,j]$ depending on some relation between $X[i]$ and $Y[j]$. Do not forget to explicitly write the base cases $M[0,j]$ and $M[i,0]$, where $1 \le i \le m, 1 \le j \le n$.

Solution:

This problem is very similar to the Longest Common Sub sequence. Big picture of our algorithm is this, we are going to find the longest common subsequence within X and Y. We will add this length to our Length Counter Z. Then we will be adding all the elemnts in X that are not in the subsequence and all the elements in Y that are not in the subsequence.

Here is how we go about doing so.

$$M[i,j] = \begin{cases} j, & \text{if } i = 0. \\ i, & \text{if } j = 0. \\ M[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } X_i = Y_j. \\ min(M[i, j-1], M[i-1, j]) + 1 & \text{if } i, j > 0 \text{ and } X_i \neq Y_j. \end{cases} \quad (1)$$

As we can see, our recurrence Equation depends on whether or not the comparing characters are the same or not. We can see this because if $X_i \neq Y_j$, there are two calls to M, where as if there is a match, there is just one.

If $X_i = Y_j$, we get the recurrence equation

```
SCS(n, m) = SCS (n - 1, m - 1) + O(1)
```

If $X_i \neq Y_j$, we get the recurrence equation

```
SCS(n, m) = SCS (n, m - 1) + SCS (n - 1, m) + O(1)
```

□

(b) (5 points) Translate this recurrence equation into an explicit bottom-up $O(mn)$ time algorithm that computes the length of the shortest common supersequence of X and Y.

Solution:

```
SCS-Length(X, Y)
{
    m = X.length
    n = Y.length
    let M[0..m, 0..n] be a new table

    for i = 1 to m
        M[i, 0] = i
    for j = 1 to n
        M[0, j] = j

    for i = 1 to m
        for j = 1 to n
            if Xi == Yj
                M[i, j] = 1 + M[i - 1, j - 1]
            elseif M[i - 1, j] <= M[i, j - 1]
                M[i, j] = 1 + M[i - 1, j]
            else
                M[i, j] = 1 + M[i, j - 1]
    return M
}
```

Here we know the algorithm is of running time $O(mn)$ because we have a nested for loop of size $M \; and \; N$, Halloween candy pun intended. But ya, we are Iterating M times, and each of these iterations is doing N Comparison operations, so we know this running time is $O(mn)$.

□

(c) (5 points) Find the SCS of $X = BARRACUDA$ and $Y = ABRACADABRA$. (Notice, you need to find the actual SCS, not only its length.)

Solution: $SCS = BABRRACAUDABRA$

In order to find this, we need to 1st find the LCS and then we need to just add the remainder elements that are not in this LCS in the correct order in regards to the LCS.

The LCS is $ARACDA$. Both elements have this, as we can see here

```
LCS = A R A C D A

X and Y with Upercase letters for where the LCS exists
    X = b A R r A C u D A
    Y = A b R A C a D A b r a

All the uppercase letters represent the letters that exist in the LCS
```

Now, we need to add all remaining elements and in the correct order. This is very similar to the merge step in a merge sort. We iterate item by item for both strings, placing element after element into our SCS. Once on of the strings hits a letter in the LCS, we will not continue placing elements from that string into our SCS until the other string has also hit that same element in the LCS. Once both strings hit the element in the LCS, we place that character into our SCS and we continue.

Here is our result

```
STEP 1
    X = b A R r A C u D A
    Y = A b R A C a D A b r a

    Y hits A initially, don't place until X hits A

    SCS = b

STEP 2
    X = A R r A C u D A
    Y = A b R A C a D A b r a

    Now Y and X both have their head pointing to the same element in the LCS, we can now p

    SCS = b A

STEP 3
    X = R r A C u D A
    Y = b R A C a D A b r a

    Now X head is at LCS element and Y isn't, so we keep popping off

    SCS = b A b

STEP Remaining

    X = R r A C u D A
    Y = R A C a D A b r a
```

```
SCS = b A b R

X = r A C u D A
Y = A C a D A b r a

SCS = b A b R  r

.......

SCS = b A b R r A C a u D A b r a
```

□

(d) (6 points) Show that the length k of the array Z computed in part (a) satisfies the equation $k = m + n - \ell$, where ℓ is the length of the longest common *subsequence* of X and Y.
(**Hint**: Use the recurrence equation in part (a), then combine it with a similar recurrence equation for the LCS, and then use induction. There the following identity is very handy: $\min(a, b) + \max(a, b) = a + b$.)

Solution:

For the SCS we see this pattern

$$SCS\ M[i,j] = \begin{cases} j, & \text{if } i = 0. \\ i, & \text{if } j = 0. \\ M[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } X_i = Y_j. \\ min(M[i,j-1], M[i-1,j]) + 1 & \text{if } i,j > 0 \text{ and } X_i \neq Y_j. \end{cases} \tag{2}$$

For the LCS we see this pattern

$$LCS\ M[i,j] = \begin{cases} 0, & \text{if } i = 0. \\ 0, & \text{if } j = 0. \\ M[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } X_i = Y_j. \\ max(M[i,j-1], M[i-1,j]) & \text{if } i,j > 0 \text{ and } X_i \neq Y_j. \end{cases} \tag{3}$$

So lets set $A = M[i, j-1]$ and $B = M[i-1, j]$ and $C = M[i-1, j-1]$

We want to prove $k = m + n - l$

$$SCS\ k = m + n - l$$

$$= (m + n)(Min(A, B) + 1) - l_{scs}(C + 1)$$

Now we replace $l_{scs}(C+1)$ with the equation for the LCS because this represents the similar elements.

$$= (m+n)(Min(A,B)+1) - (m+n)(Max(A,B)) - l(C+1)$$
$$= ((m+n)(Min(A,B)) + (m+n) - ((m+n)(Max(A,B)) - l(C+1)$$

Thanks to hint we know
$$= (m+n) - l(C+1)$$

The significance with respect to size of $l(C+1)$ is just $l(1)$ which is l.

$$= m+n-l$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms November 5, 2014

Solutions to Problem 3 of Homework 7 (18 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 5*

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 top-down recursive formula for $R[n]$ in terms of values $R[j]$ for $j < n$.

Solution:

$$R[i] = \begin{cases} 0, & \text{if } i = 0. \\ p[1] - b, & \text{if } i = 1. \\ Max(N * p[n] - B, R[j] + R[n-j]) & \text{if } i > 1, \text{ for } j = 1 \ldots \frac{n}{2}. \end{cases} \quad (4)$$

Just in case the bottom line of the recursive formula doesn't make sense, I will show an example for the last line.

For example, if $n = 7$ the last line would be

$$Max(7 * p[7] - B, R[1] + R[6], +R[2] + R[5], R[3] + R[4])$$

□

(b) (5 points) Write a top-down recursive procedure with memorization which will compute $R[n]$. Analyze the running time of your procedure in the $\Theta(\cdot)$ notation.

Solution: Code starts on next page.

```
# Setting up R Memoization table

let R[0..n] be a new table
R[0] = 0
R[1] = p[1] - B
R[2...n] = -infinity

BabyCFO(B, P, n, R)
{
    q = N * P[n] - B
    for i = 1...n
    {
        if (R[i] == -infinity)
            R[i] = BabyCFO(B, P, i, R)

        if (R[n-i] == -infinity)
            R[n-i] = BabyCFO(B, P, n-i, R)

        q = Max(q, R[i] + R[n-i])
    }
    R[n] = q
    return q
}
```

The running time of this algorithm is $\Theta(n^2)$. To solve, for each of the N elements, we must compute N different look ups. The comparison is done in $O(1)$ time, so our running time is $\Theta(n^2)$ □

(c) (5 points) Write an iterative bottom-up variant of the same procedure.

Solution: Code is on next page

```
BabyCFO(B, P, n, R)
{

    # Setting up R Initial table

    let R[0..n] be a new table
    R[0] = 0
    R[1] = p[1] - B

    for i = 2 ... n
    {
        q = n * p[i] - B
        for j = 1 ... i
        {
            q = Max(q, R[j] + R[n - j])
        }
        R[i] = q
    }
    return R[n]
}
```

This approach makes it easy to see the $\Theta(n^2)$ running time with the nested for-loop that does a Comparison and two look ups. □

(d) (3 points) Explain how to augment your procedure (either in part (b) or (c)) to also compute the optimal number of babysitters k and the actual partition of children. Either English or pseudocode will work.

Solution: Note: I have 2 code samples to show how this is done. The first shows how to set up baby sitters number and the second shows how to set up both baby sitters number and the child partition.

The optimal numbers of babysitters is very much so like the optimal Revenue. We set up a table that will be containing Baby Sitters number.
We set $B[0] = 0$, and $B[1] = 1$

Now when we check for the max, if we notice a change has happened with q, we will store the babysitter number as the sum of $R[j]$ and $R[n - j]$

Code is shown on next page.

```
BabyCFO(B, P, n, R)
{

    # Setting up R Initial table

    let R[0..n] be a new table
    R[0] = 0
    R[1] = p[1] - B

    let B-table[0..n] be a new table
    B-table[0] = 0
    B-table[1] = 1

    for i = 2 ... n
    {
        q = n * p[i] - B
        B-table[i] = i
        for j = 1 ... i
        {
            if (R[j] + R[n - j] > q)
            {
                B-table[i] = B-table[j] + B-table[n - j]
                q = R[j] + R[n - j]
            }
        }
        R[i] = q
    }
    return R[n]
}
```

To find out the partition of the babies, we do almost the exact same thing, instead the baby partition table will hold a list, and whenever that Q value is to be changed, we append the two previous lists into the list for the current element.

The code is shown on the next page.

```
BabyCFO(B, P, n, R)
{

    # Setting up R Initial table

    let R[0..n] be a new table
    R[0] = 0
    R[1] = p[1] - B

    let B-table[0..n] be a new table that stores the Baby Sitter Number
    B-table[0] = 0
    B-table[1] = 1

    let Child-P-table[0...n] be a new table of lists That stores the Child Partition
    Child-P-table[0] = 0
    Child-P-table[1] = 1

    for i = 2 ... n
    {
        q = n * p[i] - B
        B-table[i] = i
        Child-P-table[i] = 1
        for j = 1 ... i
        {
            if (R[j] + R[n - j] > q)
            {
                B-table[i] = B-table[j] + B-table[n - j]
                Child-P-table[i] = append(Child-P-table[j], Child-P-table[n - j])
                q = R[j] + R[n - j]
            }
        }
        R[i] = q
    }
    return R[n]
}
```

□

CSCI-GA.1170-001/002 Fundamental Algorithms November 5, 2014

Solutions to Problem 4 of Homework 7 (10 points)

Name: Keeyon Ebrahimi *Due: Wednesday, November 5*

You have $m \times n$ chocolate bar. You are also given a matrix $\{p[i,j] \mid 1 \le i \le m, 1 \le j \le n\}$ telling you the price of the $i \times j$ chocolate bar. You are allowed to repeat the following procedure any number of times, starting initially with the single big $m \times n$ piece you have. Take one of the pieces you have and split it into two pieces by cutting it either vertically or horizontally. Say, $m = 5$, $n = 4$. You may first choose to split it into two pieces of size 3×4 and 2×4. Then you may take the 3×4 piece and split it into two pieces 3×2 and 1×2. Finally, you may take the previous 2×4 piece and split it into two 1×4 pieces. If you stop, you have four pieces of sizes 3×2, 1×2, 1×4 and 1×4, which you can sell for $p[3,2] + p[1,2] + 2p[1,4]$. You goal is to find a partition maximizing your total profit.

(a) (5 points) Let $C[i,j]$ be the largest profit you can get by splitting an $i \times j$ piece, where $0 \le i \le m$, $0 \le j \le n$ and we set $C[i,0] = C[0,j] = 0$. Write a recursive formula for $C[m,n]$ in terms of values $C[i,j]$, where either $i < m$ or $j < n$.

Solution:
In this equation a is a loop of values from 1 to i. $a = 1...i$ and also
$b = 1...j$

$$C[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0. \\ Max(p[i,j], C[a,j] + C[i-a,j], C[i,b] + C[i,j-b]), & \text{if } i \ge 1 \text{ and } j \ge 1. \end{cases} \quad (5)$$

A pseudo code equivilant would be something like this.

```
ChocolateBar(p,i,j) {
q = p[i,j]
for a = 1....i
    q = Max(q, ChocolateBar(p,a,j) + ChocolateBar(p,i-a,j)
for b = 1...j
    q = Max(q, ChocolateBar(p,i,b) + ChocolateBar(p,i,j-b)
}
```

□

(b) (5 points) Write a bottom-up procedure to compute $C[m,n]$ and analyze its running time as a function of m and n.

Solution: Code is on next page.

```
ChocolateBar(p, m, n)
{
    let C[0...m,0...n] be a new table

    # Initialize C Table with 0 cases
    for g = 0...m
    {
        C[g,0] = 0
    }
    for h = 0...n
    {
        C[0,h] = 0
    }

    for i = 1...m
    {
        for j = 1...n
        {
            q = p[i,j]

            # For Loop A
            for a = 1 ... j
            {
                q = Max(q, C[i+a] + C[i, j-a];
            }

            # For Loop B
            for b = 1 ...i
            {
               q = Max(q, C[b,j] + C[i - b, j])
            }
            C[i, j] = q
        }
    }
    # Now entire C matrix is filled up.
    return C[m,n]
}
```

Running time: $\theta(nm(n+m))$

The bottom-up approach makes it easy to see the total run time. We iterate n times through all m, and with each iteration we are doing $m+n$ work. The $m+n$ work is the work of the for loops labeled as For Loop A and For Loop B. This means we do m times n times $m+n$ work. This equates to $\Theta(nm(n+m))$ □