Assignment 6

Ayman Shahriar UCID: 10180260 Tutorial: T02

December 3, 2020

Question 1

a) Suppose $n = 2$ and the values for l and h are given in the following table:

	week 1	week2
l	1	1
h	50	5

The optimal answer would be to pick $h_1 + l_2 = 50 + 1 = 51$ But since $h_2 > l_1 + l_2$, then the algorithm will only pick $0 + h_2 = 5$ and terminate, so the solution computed by the algorithm would not be the optimal solution.

b)
Preconditions: Two nonempty, positive integer lists $l_1, \ldots, l_n$ and $h_1, \ldots, h_n$ representing a sequence of "low stress" and "high stress" jobs respectively.

Postconditions: Return a list $s_1, \ldots, s_n$ where:

- For $1 \leq i \leq n$, s_i can be one of three things: $l, h, 0$.
- $s_i = l$ means that low stress job was selected for week i, $s_i = h$ means that high stress job was selected for week i, and $s_i = 0$ means that no job was selected for that week.
- If $s_i = h_i$, then $s_{i-1} = 0$. (It's also okay to set $s_1 = h_1$)
- The revenue generated by all the jobs in the list is maximized. So there is no other list that follows the rules above and whose total revenue from it's jobs is greater than this list.

c)
Let us define the following:
O_i = the optimal solution for input $(l_1, \ldots, l_i)$ and $(h_1, \ldots, h_i)$
OPT_i = The sum of all revenues of jobs in O_i

Assume we know $O_1, O_2, \ldots, O_{i-1}$ and $OPT_1, OPT_2, \ldots, OPT_{i-1}$.
With all of this known, how can we compute O_i?
This is will be our sub-problem.

Note that for O_i, there are two cases:
(Where $(O_i \cup n)$ means that we are appending a new element n to the end of O_i):

Case 1: The last element of O_i is "l".
Then O_i is compatible with O_{i-1}, so

$$O_i = O_{i-1} \cup \{\text{"}l\text{"}\}$$
$$OPT_i = OPT_{i-1} + l_i$$

Case 2: The last element of O_i is "h".
Then O_i is not compatible with O_{i-1}, but is compatible with O_{i-2}, so

$$O_i = O_{i-2} \cup \{\text{"}0\text{"}, \text{"}l\text{"}\}$$
$$OPT_i = OPT_{i-2} + 0 + h_i$$

(It is impossible to have a case where the last element of O_i is "0", because that would always make O_i sub-optimal)

For each instance i, the case that is true is the one that maximizes the total revenue of the solution. And to establish a base case, let us define $OPT_i = 0$ for $i < 1$.

So then:

$$OPT_i = \begin{cases} 0 \quad \text{if } i < 1 \\ max(OPT_{i-1} + l_i, \;\; OPT_{i-2} + h_i) \quad \text{otherwise} \end{cases}$$

$$O_i = \begin{cases} O_{i-1} \cup \{l\} \quad \text{if } OPT_{i-1} + l_i \geq OPT_{i-2} + h_i \\ O_{i-2} \cup \{0, h\} \quad \text{otherwise} \end{cases}$$

d)

This recursive algorithm takes in a list of low-stress and high-stress jobs, and computes the maximum value that can be earned from an optimal plan.

```
Function RecursiveMaxVal(($l_1, \ldots, l_i$), ($h_1, \ldots, h_i$))

    if $i == 0$ then
        return 0
    if $i == 1$ then
        return $max(l_1, h_1)$

    Let $v_1$ = RecursiveMaxVal(($l_1, \ldots, l_{i-1}$), ($h_1, \ldots, h_{i-1}$)) + $l_i$
    Let $v_2$ = RecursiveMaxVal(($l_1, \ldots, l_{i-2}$), ($h_1, \ldots, h_{i-2}$)) + $h_i$
    return $max(v_1, v_2)$
```

e) This recurrence relation represents an upper bound on the worst-case running time of our recursive algorithm (where c is a constant):

$T(0) = c$
$T(n) = T(n-1) + T(n-2) + c \quad$ for $n > 0$.

f) Using the guess and check method, we will prove an asymptotic upper bound on the worst case runtime of the recursive algorithm.

Using substitution, let us guess an upper bound for T(n):

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\leq T(n-1) + T(n-1) + c \quad \textbf{(since } T(n-1) \geq T(n-2) \textbf{)} \\
&= 2T(n-1) + c \\
&\leq 2(2T(n-2) + c) + c \\
&= 4T(n-2) + 3c \\
&\leq 4(2T(n-3) + c) + 3c \\
&= 8T(n-3) + 7c \\
&\ldots
\end{aligned}$$

We see a pattern: $T(n) \leq 2^k T(n-k) + (2^k - 1)c, \quad$ where $0 \leq k \leq n$.

Let k = n.
Then $T(n) \leq 2^n T(0) + (2^n - 1)c$
$= (2^n - 1)c$
$= c \cdot 2^n - c$

We will use induction on n to prove that $T(n) \leq c \cdot 2^n - c$ for all $n \geq 0$

Base Case (n = 0): $T(n) = 0 = c - c = c \cdot 2^0 - c$, as required.

Inductive Step: Suppose $T(k) \leq c \cdot 2^k - c$ for $k \geq 0$ (inductive hypothesis).

We will show that $T(k+1) \leq c \cdot 2^{k+1} - c$:

$T(k+1) = T(k) + T(k-1) + c$
$\leq T(k) + T(k) + c \quad$ (since $T(k) \geq T(k-1)$)

$\leq (c \cdot 2^k - c) + (c \cdot 2^k - c) - c$ (by the inductive hypothesis)
$= c \cdot 2^k + c \cdot 2^k - c$
$= 2 \cdot c \cdot 2^k - c$
$= c \cdot 2^{k+1} - c$, as required.

So using induction, we have proved that $T(n) \leq c \cdot 2^n - c$.

Then by definition of big-O, $T(n) = O(2^n)$.

Thus, we have proved that the worst case runtime of our recursive algorithm is in $O(2^n)$.

g)

```
Function OptimalJobs($(l_1, \ldots, l_i)$, $(h_1, \ldots, h_i)$)

    Initialize array $optRev[0, \ldots, n]$
    Initialize array $optJobs[1, \ldots, n]$

    // In order to avoid overflow of array bounds, define base cases for 0
      and 1:
    $optRev[0] = 0$
    $optJobs[0] = \{\}$

    if ($l_1 \geq h_1$): then
        $optJRev[1] = l_1$
        $optJobs[1] = \{l\}$
    else
        $optJRev[1] = h_1$
        $optJobs[1] = \{h\}$

    for $i = 2, \ldots, n$ do
        if ($optRev[i-1] + l_i \geq optRev[i-2] + h_i$): then
            $optRev[i] = optRev[i-1] + h_i$
            $optJobs[i] = optJobs[i-1] \cup \{l\}$
        else
            $optRev[i] = optRev[i-2] + h_i$
            $optJobs[i] = optJobs[i-2] \cup \{0, h\}$

    return ($optRev, optJobs$)
```

h) Let us determine the runtime of the dynamic programmming algorithm:

At the start of the algorithm, we initialize two arrays, which takes constant time. Then to avoid overflow of array bounds, we define base cases by initializing values for the arrays at index 0 and 1, which take constant time.

Let c_1 denote the constant runtime of this initial part of the algorithm.

After that, the algorithm enters a loop that iterates from 2 to n. The body of the loop consists of doing a fixed number of comparisons and accessing/updating a fixed number of arrays. Both of these take constant time, which means that each iteration of the loop runs in constant time.

Let $c_2 \cdot (n-1)$ denote the runtime of this entire loop, where c_2 is a constant and $(n-1)$ is the number of iterations.

After the loop, the algorithm returns the two arrays *oprRev* and *optJobs*, which take constant time. Let c_4 denote this constant runtime.

So the runtime function of *OptimalJobs* will be:
$c_2 \cdot (n-1) + c_5 \quad$ (where $c_5 = c_1 + c_4$)

We will use the limit test to prove that the runtime of our algorithm is in $\Theta(n)$:

$$\begin{aligned}
&\lim_{n\to\infty} \frac{c_2 \cdot (n-1) + c_5}{n} \\
&= \lim_{n\to\infty} \frac{c_2 \cdot n}{n} - \frac{c_2}{n} + \frac{c_5}{n} \\
&= c_2 \;\; \text{(which is a constant greater than 0)}
\end{aligned}$$

Thus, by definition of the limit test the runtime of our dynamic programming algorithm is in $\Theta(n)$

Question 2

Important: For this problem, we will represent each square in the checkerboard with integer pair (i, j), where i is the **column** and j is the **row**.

a) Preconditions:

- An integer $n \geq 1$ which represents the dimensions of the checkerboard. (So the checkerboard will be $n \times n$)
- A square in the checkerboard is represented by integer pairs (i, j), where i is the column and j is the row
- There are three types of moves:
  1. Move from (i, j) to $(i, j+1)$ (ie, move to the square immediately above)
  2. Move from (i, j) to $(i+1, j+1)$, as long as $i < n$ (ie, move to the square that is one up and one to the right, only if the checker is not in the rightmost column)

3. Move from (i, j) to $(i-1, j+1)$ as long as $i > 1$ (ie, move to the square that is one up and one to the left, only if the checker is not in the leftmost column)

- If you move the checker using one of the three moves without going out of bounds, then that move is legal.
- $p(x, y) \in \mathbb{R}$ is given for all pairs (x, y) for which moving a checker from squares x to y is legal.

Postconditions:

- Return a list $s_1, \ldots, s_n$ and an integer V. Each element s_k is a square location (i_k, j_k) in the checkerboard. x_k represents the column, and y_k represents the row.
- $s_1, \ldots, s_n$ represents the set of moves that will move a checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible, which is the value V. So s_1 will be a square in the botton row, and s_n will be a square in the top row.
- In other words, this list maximizes the sum of all $p(a, b)$ that can be generated from moving a checker from somewhere along the first row to somewhere along the last row. And this maximum sum is V.

b) Let $d[i, j]$ be the maximum profit earned when going from some square in row 1 to a particular square (i, j)

Then we have 3 cases:
Case 1: To get the maximum profit, we went to square (i, j) from square $(i-1, j-1)$
Then $d[i, j] = d[i-1, j-1] + p((i-1, j-1), (i, j))$

Case 2: To get the maximum profit, we went to square (i, j) from square $(i, j-1)$
Then $d[i, j] = d[i, j-1] + p((i, j-1), (i, j))$

Case 3: To get the maximum profit, we went to square (i, j) from square $(i+1, j-1)$
Then $d[i, j] = d[i+1, j-1] + p((i+1, j-1), (i, j))$

Note that some cases might not be possible depending on the position of i:
If $i = 1$, then case 2 and case 3 are only possible
If $i = n$, then case 1 and case 2 are only possible

And it is illegal to move from a square in row 1 to a square in row 1, so we can set $d[i, j] = 0$ if $j = 1$.

For other squares (i, j), the case that is true is the one that maximizes the number of dollars gathered along the way.

So we can define $d[i, j]$ to be the following:

$$d[i,j] = \begin{cases} 0 \quad \text{if } (j = 1) \\ \\ \max(d[i, j-1] + p((i, j-1), (i, j)), \\ \quad d[i+1, j-1] + p((i+1, j-1), (i, j))) \quad \text{if } (j > 1) \text{ and } (i = 1) \\ \\ \max(d[i-1, j-1] + p((i-1, j-1), (i, j)), \\ \quad d[i, j-1] + p((i, j-1), (i, j))) \quad \text{if } (j > 1) \text{ and } (i = n) \\ \\ \max(d[i-1, j-1] + p((i-1, j-1), (i, j)), \ d[i, j-1] + p((i, j-1), (i, j)), \\ \quad d[i+1, j-1] + p((i+1, j-1), (i, j))) \quad \text{if } (j > 1) \text{ and } (1 < i < n) \end{cases}$$

c) Here is a recursive algorithm that computes the maximum value that can be earned for a point (i, j):

Assume we are given $p(x, y)$ for all pairs (x, y) for which a move is legal.

```
**Function** *MaxDollars((i, j))*

    **if** $j == 1$: **then**
        **return** 0

    **if** $j > 1$ and $i == 1$: **then**
        Let
          $d_1 = MaxDollars(i, j-1) + p((i, j-1),\ (i, j))$
          $d_2 = MaxDollars(i+1, j-1) + p((i+1, j-1),\ (i, j))$
        **return** $\max(d1, d2)$

    **if** $j > 1$ and $i == n$: **then**
        Let
          $d_1 = MaxDollars(i-1, j-1) + p((i-1, j-1),\ (i, j))$
          $d_2 = MaxDollars(i, j-1) + p((i, j-1),\ (i, j))$
        **return** $\max(d1, d2)$

    **if** $j > 1$ and $1 < i < n$: **then**
        Let $d_1 = MaxDollars(i-1, j-1) + p((i-1, j-1),\ (i, j))$
          $d_2 = MaxDollars(i, j-1) + p((i, j-1),\ (i, j))$
          $d_3 = MaxDollars(i+1, j-1) + p((i+1, j-1),\ (i, j))$
        **return** $\max(\max(d1, d2),\ d_3)$
```

The non-recursive driver program computes all the values that can be earned from moving from a square in row 1 to any square in row n, and returns the highest value.

```
Function Driver(integer n)

    Let $maxVal$ = NULL
    for $i = 1, \ldots, n$: do
        Let $d = MaxDollars((i, n))$
        if $(d > maxVal)$ or $(maxVal == NULL)$: then
            $maxVal = d$
    return maxVal
```

d) When $j == 1$, the recursive algorithm makes no recursive calls. Otherwise, when $i == 1$ or $i == n$, the algorithm makes two "topmost" recursive calls. And when $1 < i < n$, the algorithm makes three "topmost" recursive calls.

So our algorithm makes at most three "topmost" recursive calls. And other than these recursive calls, all other operations (such as comparisons, return statements, etc.) run in constant time.

Note that with each recursive call, the input square gets closer to the first row by 1 row. And for any square at row 1, the algorithm terminates without making any recursive calls.

Based on the above observations, here is a recurrence relation representing an upper bound on the worst-case running time of the recursive algorithm:

$$T(n) = 3T(n-1) + c \ , \quad \text{where } c \text{ is a constant}$$

e) Using the guess and check method, we will prove an asymptotic upper bound on the worst case runtime of the recursive algorithm.

First, let us draw a recurrence tree for this recurrence relation:

Now let us find the size of the sub-problems at each level:

Level 0: $l_0 = c$

Level 1: $l_1 = 3c$

Level 2: $l_2 = 9c$

We see a pattern:

$$l_i = 3^i \cdot c$$

Note that with each recursive call, the size of the input n decreases by 1. Then for an input n, there will be a maximum of $n - 1$ levels in it's recursion tree before the recursive calls are called on input 1.

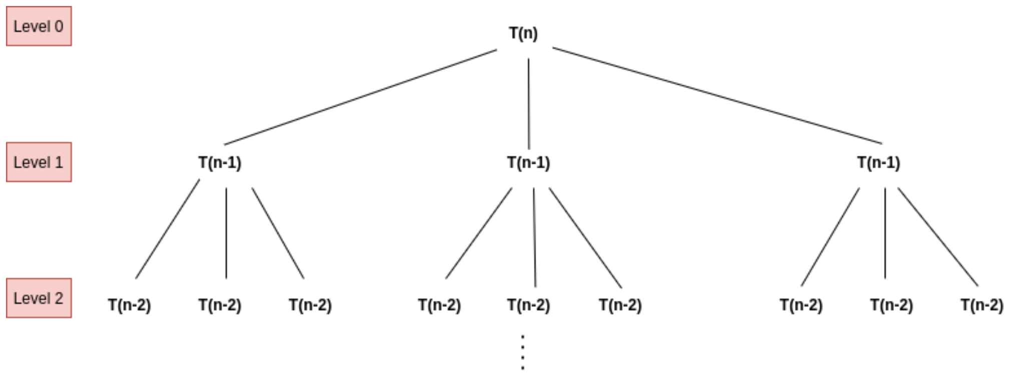

Figure 1: Recursion Tree

So taking the sum of all levels of recursion, we get:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} 3^i \cdot c \\ &\leq \sum_{i=0}^{n} 3^i \cdot c \\ &\leq \sum_{i=0}^{n} 3^n \cdot c \\ &= cn3^n \\ &\leq an3^n \quad \text{(for some } a \geq c \text{)} \end{aligned}$$

We will use the substitution method to prove that $T(n) \leq an3^n$ for some $a \geq c$:

$$\begin{aligned} T(n) &= 3T(n-1) + c \\ &\leq 3(a(n-1) \cdot 3^{n-1}) + c \\ &= a(n-1) \cdot 3^n + c \\ &\leq an \cdot 3^n \quad \text{(for a such that } an3^n \geq a(n-1)3^n + c \text{)} \end{aligned}$$

So we have shown that $T(n) \leq an3^n$ for some $a \geq c$.
Then by definition of big-O, $T(n) = O(n \cdot 3^n)$

Thus, we have found an upper bound on the runtime of our recursive algorithm.

f)

Assume we are given $p(x, y)$ for all pairs (x, y) for which a move is legal.

```
Function AllSeqVal(integer n)

    Initialize array $OPT[1 \ldots n][1 \ldots n]$
    Initialize array $Moves[1 \ldots n][1 \ldots n]$

    for $i = 1, \ldots, n$: do
        for $j = 1, \ldots, n$: do

            // The max dollars for moving from a square in row 1 to
             itself is 0
            if $j == 1$ : then
                $OPT[i][j] = 0$
                $Moves[i][j] = \{\}$

            if $j > 1$ and $i == 1$: then
                Let
                 $d_2 = OPT[i][j-1] + p((i, j-1),\ (i, j))$
                 $d_3 = OPT[i+1][j-1] + p((i+1, j-1),\ (i, j))$
                if $d_2 \geq d_3$ then
                    $Moves[i][j] = Moves[i][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_2$
                else
                    $Moves[i][j] = Moves[i+1][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_3$

            if $j > 1$ and $i == n$: then
                Let
                 $d_1 = OPT[i-1][j-1] + p((i-1, j-1),\ (i, j))$
                 $d_2 = OPT[i][j-1] + p((i, j-1),\ (i, j))$
                if $d_1 \geq d_2$: then
                    $Moves[i][j] = Moves[i-1][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_1$
                else
                    $Moves[i][j] = Moves[i][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_2$

            if $j > 1$ and $1 < i < n$: then
                Let
                 $d_1 = OPT[i-1][j-1] + p((i-1, j-1),\ (i, j))$
                 $d_2 = OPT[i][j-1] + p((i, j-1),\ (i, j))$
                 $d_3 = OPT[i+1][j-1] + p((i+1, j-1),\ (i, j))$
                if $d_1 \geq d_2$ and $d_1 \geq d_3$: then
                    $Moves[i][j] = Moves[i-1][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_1$
                if $d_2 \geq d_1$ and $d_2 \geq d_3$: then
                    $Moves[i][j] = Moves[i][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_2$
                else
                    $Moves[i][j] = Moves[i+1][j-1] \cup \{$(i, j)$\}$
                    $OPT[i][j] = d_3$

    return ($OPT$, $Moves$)
```

So after calling $AllSeqVal(n)$, the array $OPT[i][j]$ is filled with the maximum possible value earned from moving from some square in row 1 to the square at (i, j). And the array $Moves[i][j]$ is filled with the sequence of moves made to get from some square in row 1 to the square at (i, j) and earn the maximum amount of dollars.

So in order to find out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible, we simply need to see which square (i, j) along the top row has the highest OPT value, and return $Moves[i][j]$.

So after running $AllSeqVal(n)$, we will call the following function:

```
Function OptimumSequence(int n)

    Let $optVal$ = NULL
    Let $optSeq$ = {}

    for $i = 1, \ldots, n$: do
        if ($OPT[i][n] > optVal$) or ($optVal == NULL$): then
            $optVal = OPT[i][n]$
            $optSeq = Moves[i][n]$

    return ($optSeq$, $optVal$)
```

g) Let us determine the runtime of the $AllSeqVal$ algorithm: At the start of the function $AllSeqVal(n)$, we initialize two lists OPT and $Moves$, which takes constant time. Let c_1 denote this constant time.

After that, the algorithm enters a nested loop, where each loop iterates from 1 to n (so the nested loop iterates n^2 times). Inside this nested loop, all the statements consist of either doing comparisons, or accessing and updating an array location. Since both of these take constant time, it means that each iteration of the nested loop takes constant time.

Let $c_2 \cdot n^2$ denote the runtime of the entire nested loop.

After the nested loop, the algorithm returns the arrays OPT and $Moves$, which takes constant time. Let c_3 denote this runtime.

So the runtime function of $AllSeqVal$ is:

$$c_2 \cdot n^2 + c \quad \text{(where } c = c_1 + c_2\text{)}$$

Now, let us determine the runtime of the $OptimumSequence$ algorithm: Initializing arrays $optVal$ and $optSeq$ take constant time. Let c_4 denote this constant.

After that, the algorithm iterates n times, where each iteration consists of one comparison statement and two array access/update statements, all of which take constant time. Let $c_5 \cdot n$ denote the runtime of the loop.

At the end, the algorithm returns *optSeq*, which takes constant time. Let c_6 denote this constant time.

Then the runtime function of *optSeq* is:

$$c_5 \cdot n + c_7 \quad \text{(where } c_7 = c_4 + c_6\text{)}$$

So the runtime of our entire dynamic programming algorithm (where $AllSeqVal$ is first executed and then $OptimumSequence$ is executed) is:

$$c_2 \cdot n^2 + c_5 \cdot n + c_8 \quad \text{(where } c_8 = c_7 + c\text{)}$$

We will now use the limit test to prove that our algorithm is in $\Theta(n^2)$:

$$\begin{aligned}
&\lim_{n\to\infty} \frac{c_2 \cdot n^2 + c_5 \cdot n + c_8}{n^2} \\
&= \lim_{n\to\infty} \frac{c_2 \cdot n^2}{n^2} + \frac{c_5 \cdot n}{n^2} + \frac{c_8}{n^2} \\
&= \lim_{n\to\infty} c_2 + \frac{c_5}{n} + \frac{c_8}{n^2} \\
&= c_2 \quad \text{(which is a constant greater than 0)}
\end{aligned}$$

Thus, the runtime of our dynamic programming algorithm is in $\Theta(n^2)$.