CS 6363.003 Spring 2021	Midterm 1 Problem 1 Solution

Please answer each of the following questions. You do not have to justify your answers.

(a) **(2.5 out of 10)** Using Θ-notation in terms of n, what is the solution to the recurrence $T(n) = 9T(n/3) + n^2$?

Solution: Each level of the recursion tree sums to $\Theta(n^2)$, and there are $\Theta(\log n)$ levels. Therefore, $\boldsymbol{T(n) = \Theta(n^2 \log n)}$**.** ■

(b) **(2.5 out of 10)** Using Θ-notation in terms of n, what is the solution to the recurrence $T(n) = T(4n/7) + T(2n/7) + n$?

Solution: Each row of the recursion tree sums to *at most* $6n/7$. The row sums add up to something growing no faster than a decreasing geometric series which is bounded by its largest term. Therefore, $\boldsymbol{T(n) = \Theta(n)}$**.** ■

(c) **(2.5 out of 10)** Using Θ-notation in terms of n, what is the solution to the recurrence $T(n) = 5T(n/3) + n$?

Solution: Level i of the recursion tree sums to $(5/3)^i n$. The row sums form an increasing geometric series which is asymptotically equal to its largest term, the number of leaves. Therefore, $\boldsymbol{T(n) = \Theta(n^{\log_3 5})}$**.** ■

(d) **(2.5 out of 10)** Consider the following recursive function which is defined in terms of a fixed array $X[1 .. n]$.

$$LaLa(i,j) = \begin{cases} 0 & \text{if } i \le 0 \text{ or } j \le 0 \\ \max_{1 \le k \le i}(X[i] \cdot LaLa(i-k, j-1)) & \text{otherwise} \end{cases}$$

Using Θ-notation in terms of n, how long does it take to compute $LaLa(n,n)$ using dynamic programming?

Solution: For all subproblems we may solve, $0 \le i \le n$, and $0 \le j \le n$. Therefore, there are $\Theta(n^2)$ subproblems. All subproblems take $O(n)$ time to solve given their dependencies already have solutions, and most subproblems take $\Omega(n)$ time to solve, so **the running time is $\boldsymbol{\Theta(n^3)}$.** ■

CS 6363.003 Spring 2021 — Midterm 1 Problem 2 Solution

(a) **(5 out of 10)** Let $SCS(i,j)$ be the length of the shortest common supersequence of $A[1\,..\,i]$ and $B[1\,..\,j]$. Rewrite the following recursive definition of $SCS(i,j)$ with all ***five*** blanks filled in correctly. There is no need to justify your answer.

Solution:

$$SCS(i,j) = \begin{cases} i & \text{if } j = 0 \\ \underline{\boldsymbol{j}} & \text{if } i = 0 \\ \min\begin{Bmatrix} SCS(i,j-1)+\underline{\mathbf{1}} \\ SCS(i-1,j)+\underline{\mathbf{1}} \end{Bmatrix} & \text{if } A[i] \neq B[j] \\ SCS(\underline{\boldsymbol{i-1}}\,,\ \underline{\boldsymbol{j-1}}) + 1 & \text{otherwise} \end{cases}$$

Explanation: If either sequence is empty, then the shortest common supersequence is the other sequence. Therefore, $SCS(i,0) = i$ and $SCS(0,j) = j$. Now, suppose $i > 0$ and $j > 0$. Let $C[1\,..\,k]$ be a shortest common supersequence of $A[1\,..\,i]$ and $B[1\,..\,j]$. Any common supersequence ending with a character other than $A[i]$ or $B[j]$ can be shortened by removing its last character, so $C[k] = A[i]$ or $C[k] = B[j]$.

Suppose $A[i] \neq B[j]$. Further, suppose $C[k] = A[i]$. Then, we know $C[1\,..\,k-1]$ is a supersequence of $A[1\,..\,i-1]$, and it is also a supersequence for all of $B[1\,..\,j]$ as well. In fact, it is in fact the shortest such sequence. Counting the one character $C[k]$ in the length, we have $SCS(i,j) = SCS(i-1,j)+1$ for this case. Of course, it could be that $C[k] = B[j]$ instead. A symmetric argument implies $SCS(i,j) = SCS(i,j-1)+1$ for this case. We don't know in advance which of $C[k] = A[i]$ or $C[k] = B[j]$ is true, so $SCS(i,j)$ must be the minimum of the two possibilities.

Finally, suppose $A[i] = B[j]$. All we need is for $C[1\,..\,k-1]$ to be a common supersequence of both $A[1\,..\,i-1]$ and $B[1\,..\,j-1]$, and it should again be the shortest such supersequence. Again, counting the character $C[k]$ in the length, we have $SCS(i,j) = SCS(i-1,j-1)+1$. ■

(b) **(5 out of 10)** You're given an array $L[1\,..\,n]$ of *lumbers* and a procedure MERGE3($W[1\,..\,k]$) that sorts a subarray of lumbers $W[1\,..\,k]$ in $O(k)$ time assuming the subarrays $W[1\,..\,\lfloor k/3\rfloor]$, $W[\lfloor k/3\rfloor+1\,..\,\lfloor 2k/3\rfloor]$, and $W[\lfloor 2k/3\rfloor+1\,..\,k]$ are already sorted. Design *and analyze* an algorithm to sort the array L. Accessing elements of and making change to L must occur only within calls to MERGE3. In particular, you ***may not*** compare lumbers directly.

You do not need to justify correctness of your algorithm for this problem.

Solution: The divide-and-conquer procedure SORTLUMBERS($L[1\,..\,n]$) sorts the given array L of lumbers per the problem's requirements.

```
SORTLUMBERS($L[1 .. n]$):
  if $n > 1$
      SORTLUMBERS($L[1 .. \lfloor n/3\rfloor]$)
      SORTLUMBERS($L[\lfloor n/3\rfloor + 1 .. \lfloor 2n/3\rfloor]$)
      SORTLUMBERS($L[\lfloor 2n/3\rfloor + 1 .. n]$)
      MERGE3($L[1 .. n]$)
```

The procedure SortLumbers($L[1 \,..\, n]$) performs 3 recursive calls on subproblems of size approximately $n/3$. The call to Merge3 takes $O(n)$ time. Therefore, the running time of SortLumbers($L[1 \,..\, n]$) follows the recurrence $T(n) = 3T(n/3) + O(n)$. Each level of the recursion tree for this recurrence sums to at most cn for some constant c, and there are $O(\log n)$ levels. **The running time is $O(n \log n)$.**

Explanation: If $n \leq 1$, the array $L[1 \,..\, n]$ is already sorted, and there is nothing to do. Otherwise, we perform recursive calls on the three subarrays that a call to Merge3($L[1 \,..\, n]$) would expect to be sorted. Each recursive call takes a subarray of length strictly less than n, so induction guarantees the calls successfully sort their inputs. The call to Merge3 then sorts L as promised. ■

CS 6363.003 Spring 2021 Midterm 1 Problem 3 Solution

(10 points) Suppose you are given an array $A[1\,..\,n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a ***local minimum*** if it is less than or equal to both of its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$.

Describe and analyze a faster algorithm that finds a local minimum in $O(\log n)$ time. You should *briefly* justify the correctness and running time of your algorithm.

Solution: The divide-and-conquer/binary search procedure LocalMin($A[1\,..\,n]$) takes an array $A[1\,..\,n]$ with the special property described in the problem and returns a single element that is a local minimum. To avoid ambiguity with the problem description, we'll assume $n \geq 3$. Any reasonable interpretation of what to do when $n < 3$ is still worth full credit.

```
LocalMin($A[1 .. n]$):
  $m \leftarrow \lceil n/2 \rceil$
  if $A[m-1] < A[m]$
      return LocalMin($A[1 .. m]$)
  else if $A[m] > A[m+1]$
      return LocalMin($A[m .. n]$)
  else
      return $A[m]$
```

Our assumption that $n \geq 3$ implies $2 \leq m \leq n-1$. If $A[m-1] < A[m]$, then the subarray $A[1\,..\,m]$ has the property that $A[1] \geq A[2]$ and $A[m-1] \leq A[m]$. Further, we may infer $3 \leq m \leq n-1$. By induction, the call LocalMin($A[1\,..\,m]$) returns a local minimum. Similarly, if $A[m] > A[m+1]$, then $A[m\,..\,n]$ also has the boundary property and a length between 3 and $n-1$. By induction, the call LocalMin($A[m\,..\,n]$) returns a local minimum. If both inequalities are false, then $A[m]$ itself is a local minimum that we can return. Note that we do not need to explicitly consider an extra base case, because we always directly return the element $A[m]$ in situations where we are not guaranteed a successful recursive call.

The algorithm does at most one recursive call on a subproblem of about half the input size. The algorithm therefore follows the running time recurrence $T(n) \leq T(n/2) + O(1)$ which solves to $O(\log n)$. ■

CS 6363.003 Spring 2021 Midterm 1 Problem 4 Solution

Suppose you are given an array $P[1 .. n]$ where $P[i]$ denotes the amount of revenue you get from selling a Flower Monster i days after first planting it. You want to design an algorithm that will tell you the maximum total profit (total revenue minus the total cost of planting new Flower Monsters) you can obtain over the course of n days.

(a) **(5 out of 10)** For all $0 \le i \le n-1$, let $MaxProfit(i)$ be the maximum total profit you can obtain if you plant a new Flower Monster on day i and proceed to sell and plant Flower Monsters over the course of days $i+1$ through n. Our ultimate goal is to compute $MaxProfit(0)$.

Give a recursive definition of $MaxProfit(i)$. If you wish, you may propose a convinient definition for $MaxProfit(n)$ to use as a base case. You should *briefly* justify the correctness of your recursive definition.

Solution: We will base our recursive definition on the following idea: Because we're planting a new Flower Monster on day i, we should guess which day j we should sell it. We have to wait at least one day to sell, and we must sell some Flower Monster on day n, so $i+1 \le j \le n$. Buying the Flower Monster on day i costs 10, but we'll immediately receive $P[j-i]$ from the selling on day j.

Now, if $j \ne n$, then we'll immediately plant a new Flower Monster. We should do the most profitable things we can on days $j+1$ through n, leading to a total profit of $P[j-i] - 10 + MaxProfit(j)$.

If we choose to sell on day n, however, we do not plant any more Flower Monsters. For simplicity, we'll define $MaxProfit(n) := 0$ to reflect the lack of business on days n and beyond.

The above discussion leads to the following recursive definition.

$$MaxProfit(i) = \begin{cases} 0 & \text{if } i = n \\ \max_{i+1 \le j \le n} (P[j-i] - 10 + MaxProfit(j)) & \text{otherwise} \end{cases}$$

■

(b) **(5 out of 10)** Describe and analyze an efficient dynamic programming algorithm to compute the maximum total profit you can obtain over the course of n days. You do not need to justify correctness of your algorithm, but you should go through the standard memoization steps anyway to make sure your algorithm is correct.

Solution: As stated in the problem description for part (a), our ultimate goal is to compute and return $MaxProfit(0)$. For our subproblems, we'll have $0 \le i \le n$. Therefore, we can store the solutions in an array $MaxProfit[0 .. n]$. Each entry depends upon others with strictly larger index, so we'll fill the array in decreasing order of index, i.e., from right to left. Each of the $O(n)$ entries takes $O(n)$ time to compute given its dependencies, because we have to loop over all the choices of j. Therefore, **the running time will be $O(n^2)$**.

Here's the pseudocode.

```
FlowerMonsterProfit($P[1 .. n]$):
    $MaxProfit[n] \leftarrow 0$
    for $i \leftarrow n-1$ down to 0
        $MaxProfit[i] \leftarrow -\infty$
        for $j \leftarrow i+1$ to $n$
            if $P[j-i] - 10 + MaxProfit[j] > MaxProfit[i]$
                $MaxProfit[i] \leftarrow P[j-i] - 10 + MaxProfit[j]$

    return $MaxProfit[0]$
```

■