HW #1

Ryan St.Pierre

September 11, 2017

Problem 1A

$$T(n) = T(\frac{n}{2}) + 2T(\frac{n}{4}) + n^2$$

I originally thought about this problem using the recursive tree approach. The first layer has 1 block of n size. The second layer has 1 block of $\frac{n}{2}$ size and 2 blocks of $\frac{n}{4}$ size. The third layer has 1 block of $\frac{n}{2}$ size, 4 blocks of $\frac{n}{8}$ size, and 4 blocks of $\frac{n}{16}$ size. This pattern continues. However, as seen below, I only needed three layers of the recursion tree to establish the necessary pattern for merge work per layer.

For the sake of clarity the recurrence relation at each level of the recursive tree is given below - where at each level I note the merge work done at each layer and carry through the work not accounted for to the next level.

Layer 1 (i=0)

$T(\frac{n}{2}) + 2T(\frac{n}{4}) + n^2$
Layer (additional) merge cost: n^2

Layer 2 (i=1)

$T(\frac{n}{2}) + 2T(\frac{n}{4}) = T(\frac{n}{4}) + 2T(\frac{n}{8}) + (\frac{n}{2})^2 + 2(T(\frac{n}{8}) + 2T(\frac{n}{16}) + (\frac{n}{4})^2)$
$= T(\frac{n}{4}) + 4T(\frac{n}{8}) + 4T(\frac{n}{16}) + (\frac{3}{8})n^2$
Layer (additional) merge cost: $(\frac{3}{8})n^2$

Layer 3 (i=2)

$T(\frac{n}{4}) + 4T(\frac{n}{8}) + 4T(\frac{n}{16}) = T(\frac{n}{8}) + 2T(\frac{n}{16}) + (\frac{n}{4})^2 + 4(T(\frac{n}{16}) + 2T(\frac{n}{32}) + (\frac{n}{8})^2) + 4(T(\frac{n}{32}) + 2T(\frac{n}{32}) + (\frac{n}{16})^2)$
$= T(\frac{n}{8}) + 6T(\frac{n}{16}) + 12T(\frac{n}{32}) + 8T(\frac{n}{64}) + (\frac{9}{64})n^2$
Layer (additional) merge cost: $(\frac{9}{64})n^2$

At this point we will hypothesize that the additional merge work per layer is equal to $\frac{3^i}{8^i}n^2$, which is satisfied by layers 1-3. We will use this to find our running time and prove its correctness by showing it satisfies the recurrence relation for all n. First, we must establish the number of layers in the recursive tree. The recursive tree is not balanced. Its longest length is $\log_2 n$, which we will use to approximate the running time. Since we are looking at asymptotic behavior this approximation should have no effect on our final result.

$$\begin{aligned}
T(n) &= \textstyle\sum_{i=1}^{\#oflayers} \text{merge cost of layer}_i \\
&= \textstyle\sum_{i=1}^{\log_2 n} \frac{3^i}{8^i} n^2 \\
&= n^2 \textstyle\sum_{i=1}^{\log_2 n+1} (\frac{3}{8})^i && \text{geometric series} \\
&= n^2 \frac{1-(\frac{3}{8})^{\log_2 n}}{1-\frac{3}{8}}
\end{aligned}$$

As n approaches ∞ the expression $n^2 \frac{1-(\frac{3}{8})^{\log_2 n+1}}{1-\frac{3}{8}}$ approaches $\frac{8}{5}n^2$. Therefore, $T(n) = \frac{8}{5}n^2$ for large n and $T(n) = \Theta(n^2)$.

$$\mathbf{T(n) = \Theta(n^2)}$$

The found running time is shown to satisfy the recurrence relation below using induction:

Claim: for $n \geq 4$, $T(n) = \frac{8}{5}n^2$
Base case: n=4

Let T(2) = A and T(1) = B, where A and B are constants

$$\begin{aligned}
T(4) &= T(2) + 2T(1) + n^2 \\
T(4) &= A + 2B + n^2
\end{aligned}$$

It is clear that a C can be chosen such that $CT(4) > \frac{n}{4}$ and $CT(4) < \frac{n}{4}$, thus proving $T(4) = \Theta(n^2)$ (base case).

Hypothesis: $T(\frac{n}{2}) + 2T(\frac{n}{4}) + n^2$ for $4 \leq n \leq k$

$$\begin{aligned}
T(k) &= T(\tfrac{k}{2}) + 2T(\tfrac{k}{4}) + k^2 && \text{from recursion} \\
\tfrac{8}{5}k^2 &= \tfrac{8}{5}(\tfrac{k}{2})^2 + 2\tfrac{8}{5}(\tfrac{k}{4})^2 + k^2 && \text{from induction hypothesis} \\
\tfrac{8}{5}k^2 &= \tfrac{2}{5}k^2 + \tfrac{1}{5}k^2 + k^2 \\
\tfrac{8}{5}k^2 &= \tfrac{8}{5}k^2
\end{aligned}$$

Problem 1B

$$T(n) = n^{1/3}T(n^{2/3}) + n$$

To analyze the running time of this problem I will start by finding the merge cost of the top layer. I will then find the running time expression for the succeeding layer, ignoring the merge cost of the layer above which has already been accounted. I will then find the merge cost of this layer, repeating this process until I find a relationship between the layer and merge cost. This approach is similar to the recursive tree approach - but I find my work semantically easier to follow than the recursion tree.

Layer 1 (i=0)

$T(n) = n^{1/3}T(n^{2/3}) + n$
Layer (additional) merge cost: n

Layer 2 (i=1)

$n^{1/3}T(n^{2/3}) = n^{1/3}(n^{2/9}T(n^{4/9}) + n^{2/3})$
$n^{1/3}T(n^{2/3}) = n^{5/9}(n^{4/9}) + n$
Layer (additional) merge cost: n

Layer 3 (i=2)

$n^{5/9}T(n^{4/9}) = n^{5/9}(n^{4/27}T(n^{8/27}) + n^{4/9})$
$n^{5/9}T(n^{4/9}) = n^{19/27}T(n^{8/27}) + n$
Layer (additional) merge cost: n

At this point we can guess that the merge cost per layer is n. We will use this guess to find the running time and then verify its correctness. However, first we must find the number of layers (analogous to the length of the recursive tree). We find that for each increase in layer n maps to $n^{2/3}$ (a non-linear relationship). We will contrive a system below in which q is defined in such a way that it decrements by one every layer. We will then find q in terms of the n to get the length of the tree.

$$\text{Let } n = 2^p, \text{ thus } p = \log_2 n$$

Let n' denote the size of each problem in the $i + 1$ layer with respect to layer i, where the size of the problem is n. Let p and q also follow this notation. From this notation it follows:

$$n' = n^{2/3} \text{ thus } p' = \frac{2}{3}p$$

Next, define q such that $p = (3/2)^q$, meaning $q = \log_{3/2} p$
Using these definitions it is shown below that $q' = q - 1$

$$p' = \frac{2}{3}p$$
$$p' = \frac{2}{3}(\frac{3}{2})^q$$
$$p' = (\frac{3}{2})^(q-1)$$

Finally, we find q, the linearly decremented variable, in terms of n.
$p = \log_2 n$ and $q = \log_{3/2} p$, therefore $q = \log_{3/2} \log_2 n$

This means the depth of the recursive tree is $\log_{3/2} \log_2 n$

Now, given the recursive tree depth, the running time can be analyzed as follows.

$$\begin{aligned}
T(n) &= \textstyle\sum_{i=1}^{\#oflayers} \text{merge cost of layer}_i \\
&= \textstyle\sum_{i=1}^{\log_{3/2} \log_2 n} n \\
&= n \textstyle\sum_{i=1}^{\log_{3/2} \log_2 n} 1 \\
T(n) &= n \log_{3/2} \log_2 n
\end{aligned}$$

$$\mathbf{T(n) = \Theta(n \log \log n)}$$

The verification that $T(n) = \log_{3/2} \log_2 n$ satisfied the recurrence relation is given below.

$$\begin{aligned}
T(n) &= n^{1/3} T(n^{2/3}) + n \\
&= n^{\frac{1}{3}} T(n^{\frac{2}{3}}) + n \\
&= n^{\frac{1}{3}} (n^{\frac{2}{3}} \log_{3/2} \log_2 n^{\frac{2}{3}}) + n \\
&= n \log_{3/2} \log_2 n^{\frac{2}{3}} + n \\
&= n(\log_{3/2} \log_2 n^{\frac{2}{3}} + 1) \\
&= n(\log_{3/2} (\frac{2}{3} \log_2 n) + 1) \\
&= n(\log_{3/2} (\frac{2}{3}) + \log_{3/2} \log_2 n + 1) \\
&= n(-1 + \log_{3/2} \log_2 n + 1) \\
T(n) &= n \log_{3/2} \log_2 n
\end{aligned}$$

Problem 2A

The binary search algorithm divides the problem in half every time the method is called and the desired number is not found. Additionally, the binary search method does an equality check each time it is called, which means each method call is doing constant work. The one time the method does constant work slightly different than all other cases is when the length of the input list is less than 3. However, since this case also does constant work (check 2 numbers) and we care about asymptotic behavior we can assume that each layer does constant work A. Given the information above the following recurrence relation can be defined for the binary search algorithm,

$$T(n) = T(\frac{n}{2}) + A$$

Note: this recurrence relation describes the worst case scenario of the algorithm, in which the problem is broken down until 2 numbers remain. The best case scenario is when the desired number is in the center of the original list. In this case the performance of the algorithm is O(1).

Using the recursive tree method to analyze the recurrence relation:

Layer 1

- 1 block of size n with constant work A

Layer 2

- 1 block of size $\frac{n}{2}$ with constant work A

Layer 3

- 1 block of size $\frac{n}{4}$ with constant work A

...

Layer i

- 1 block of size $\frac{n}{i^2}$ with constant work A

...

Layer $\log_2 n$

- 1 block of size 2 with constant work A

Since the problem is divided in two every time the length of the recursive tree is given by $\log_2 n$. The running time can be analyzed as follows.

$$\begin{aligned}
T(n) &= \textstyle\sum_{i=1}^{\#oflayers} \text{merge cost of layer}_i \\
&= \textstyle\sum_{i=1}^{\log_2 n} A \\
&= A \textstyle\sum_{i=1}^{\log_2 n} 1 \\
T(n) &= A \log_2 n
\end{aligned}$$

$$T(n) = A \log_2 n$$

$$\mathbf{T(n) = O(log\ n)}$$

Again, this is given as big 0, not Θ because this is an upper bound for the algorithm. In the best case (when the desired value is in the middle of the list) the running time of the algorithm is constant.

Problem 2B

Like the binary search the algorithm can begin by checking the middle element of the array A (at location k) and its adjacent neighbors to the left and the right. Based on this check several different actions should be taken, detailed below:

1. If A[k-1] > A[k] and A[k] > A[k+1] index k is in the decreasing section of the list. The smallest element must be to the right. Partition the list to the right starting at k.
2. If A[k-1] < A[k] and A[k] < A[k+1] index k is in the increasing section of the list. The smallest element must be to the left. Partition the list to the left starting at k.
3. If A[k-1] > A[k] and A[k] < A[k+1] then the element at index k must be the smallest element. Return A[k]. This can be considered the base case.

Note: A[k-1] < A[k] and A[k] > A[k+1] if not a valid case given how the list has been described in the problem.

The psudo-code for the algorithm is given below:

```
smallest_element(A[1...n])
    let k = n/2
    Partition A into B, C
        where B contains A[1...k-1] and C contains A[k+1...n]

    if A[k-1]>A[k] and A[k] > A[k+1]
        then return smallest_element(C)  // go right
    if A[k-1]<A[k] and A[k] < A[k+1]
        then return smallest_element(B)  // go left

    // if this executes then A[k-1] > A[k] and A[k] < A[k+1]
    return A[k]
```

Note: For the sake of simplicity this psudo-code ignores edge cases, such as when p=1 or p=n. All equality checks above need to be altered to check out of bounds if these edge cases are to be considered.

Proof of correctness

In this problem we are given an array A[1...n] that is first decreasing and then increasing. In other words there exists an index p such that $1 \leq < p \leq n such that for i < p$ and for all $i < p$, A[i] > A[i+1] and for all $i \geq p$, A[i] < A[i+1].

First note if A[1...a, b...n] where A[p] = b then the smallest element is b. This falls directly from the problem statement. Element a is in the decreasing portion of the list meaning all elements to the left of a are greater than a. Element b falls in the increasing portion of the list meaning all elements to the right of b are greater than b. Additionally, a falls at the p-1 index, meaning A[p-1] > A[p] or a>b must be satisfied. If all elements from 1...p-2 are greater than the element at p-1, all elements from p+1...n are greater than the element at p, and the element p-1 is greater than the element at p it must follow that the element at p is the smallest in the array. Thus, from this point it is now the sole burden of proof of correctness is proving the given algorithm approaches element p. To do such it will be shown that in all cases the algorithm partitions the array to approach element at location p until the base case is reached.

Let B be the current partition of the array A and let k be the middle element of partition B. There are 3 cases that follow:

1. *(k=p)*. In this case the algorithm should return A[p]. The algorithm will return such if A[k+1] > A[k] and A[k-1] > A[k]. This is satisfied be the problem statement, as A[p-1] > A[p] and A[p+1] > A[p]- thus the proper element is returned when k=p.

2. *(k<p)*. In this case index k falls in the decreasing portion of the array an the algorithm should partition to the right. The algorithm will partition to the right if A[k-1] > A[k] and A[k] > A[k+1], which again follows from the proper statement. Thus, the algorithm will properly partition to the right and approach the base case (Case 1).

3. *(k>p)*. In this case index k falls in the increasing portion of the array an the algorithm should partition to the left. The algorithm will partition to the left if A[k-1] < A[k] and A[k] < A[k+1], which again follows from the proper statement. Thus, the algorithm will properly partition to the left and approach the base case (Case 1).

Running Time

Finally, it can be shown that the above algorithm has the same running time as the binary search. Just like binary search the above "smallest element" algorithm can be described with the recurrence relation:

$$T(n) = T(\frac{n}{2}) + A$$

The solution to this recurrence relation is given by $O(\log n)$. This can be seen by evaluating the recurrence relation with a recursion tree, where the are $\log_2 n$ layers and constant work per level. Each level i has one block of size $n/2^i$. Again this running time is given as big O rather than big Θ because the best case for the "smallest element" algorithm is constant in the case where the smallest element is in the middle of the list. The small inductive proof that this designed algorithm has a running time of O(log n) is given below:

Given: the algorithm satisfies the recurrence relation, $T(n) = T(\frac{n}{2})$ + A, where A is a positive constant and T(2) = 1
Claim: for $n \geq 2$, $T(n) \leq \log_2 n$
Hypothesis: $T(n) \leq \log_2 n for 2 \leq n \leq k$
Base case: for n = 2 $T(2) = T(1) + A = 1 + A \leq log_2 2 = 1$

Proof.

$$\begin{aligned}
T(k) &= T(\tfrac{k}{2}) + A && \text{from recursion} \\
T(k) &\leq \log_2 \tfrac{k}{2} + A && \text{induction hypothesis} \\
&= \log_2 k - \log_2 2 + A \\
&= \log_2 k - 1 + A \\
&\leq \log_2 k + A
\end{aligned}$$

Problem 3

The following is the implementation of the algorithm garden(grid) where grid is the nxn garden. Let n in the following implementation be the length (or width) or the grid based into the garden method, meaning in the first call $n = k$.

1. If the grid (block of garden) has a length (and width) equal to 2 place a block in a proper orientation and **return**. In this base case 1 of the 4 blocks should already be occupied, thus there is only one proper way to orient the tile to fit.

2. If the grid (block of garden) has a length (and width) greater than 2 place a block in the partition the k by k block of the garden into 4 uniform squares A, B, C, D of size $\frac{k^2}{4}$ where A has the tiles in the top left (0 $\rightarrow$ k/2, 0 $\rightarrow$ k/2), B has the tiles in the top right (k/2 $\rightarrow$ k, 0 $\rightarrow$ k/2), C has the tiles in the bottom left (k/2 $\rightarrow$ k, 0 $\rightarrow$ k/2), and D has the titles in the bottom right (n/2 $\rightarrow$ k, k/2 $\rightarrow$ k).

3. Place a tile in the middle of the passed in grid, at $(\frac{k}{2}, \frac{k}{2})$. Its orientation should be in a way such that after its placed A, B, C, and D each have one space occupied (either by a tile or tree)

4. Recurse - calling grid(A), grid(B), grid(C), and grid(D).

Psudo-code is given below to make the described steps above more clear.

```
fill_garden(A[1...n])
```

```
if area of A = 4
    then place tile in 3 empty spaces and RETURN

Partition A into B, C, D, E where each have area equal to
the area of a over 4 such that B corresponds to the
top left corner, C the top right, D the bottom left,
and E the bottom left

Place a tile in the center of A in an orientation such that
after placement B, C, D, and E each have exactly one occupied
space (either a tree or tile)

fill_garden(B)
fill_garden(C)
fill_garden(D)
fill_garden(E)
```

Proof of correctness

Problem statement: Let A be an n by n garden with one location occupied. Given the problem statement n is a power of 2.

Base case: n = 2 (area equals 4)
In this case the tile can be oriented in a way to avoid the one occupied location, regardless of its location in A.

Let k be $\frac{n}{2}$. Since n is a power of 2, k must also be a power of 2. Let B, C, D, and E be $\frac{k}{2}$ by $\frac{k}{2}$ partitions of A corresponding to the four corners of A. Given A only has one occupied location, only one of the partitions can have an occupied location. Said differently, three of the partitions will be completely free. Thus, since the tile is an L with 3 tiles, it can be placed in the center of A with a tile in the three free partition, creating four partitions each with one occupied space. In this manner we have recreated the problem statement four times over where $n = \frac{k}{2}$ and k still satisfies the necessary requirements of n (that it is a power of 2).
To complete the proof of correctness it needs to be shown that the given algorithm approaches the base case. Let l be the length of the garden. The

base case is reached when l = 2. The original garden has length n, where n is a power of 2. Each time the algorithm divides n by 2, thus creating a smaller power of 2. Eventually, successive divisions of a power of 2 by 2 will produce 2, the base case. More clearly, if the starting garden size (where size corresponds to the length) is $n = 2^i$ it will take i iterations of the algorithm to produce a partition of the garden of size 2. The same argument holds for the width, given the symmetry of the problem.

Running time analysis

Each recursion the algorithm does constant work (placing a tile) and divides the problem into four problems of a quarter the size. This can be summarized by the recurrence relation,

$$T(n) = 4T(\frac{n}{4}) + A$$

This recurrence relation can be described by a recursion tree with constant work at each layer and a height of $\log_4 n$

Recursion tree

Layer 1 (i=0)

One block of size n
Layer cost: A

Layer 2 (i=1)

4 blocks of size $\frac{n}{4}$
Layer cost: A

Layer 3 (i=2)

16 blocks of size $\frac{n}{16}$
Layer cost: A

...
Layer i

4^i blocks of size $\frac{n}{4^i}$
Layer cost: A

...

Last layer

$\frac{n}{4}$ blocks of size 4
Layer cost: A

This recursion tree has $\log_4 n$ layers, each with constant work A.

$$\log_4 n = \Theta(\text{log n})$$