Analysis of Algorithms Midterm Exam

Philip Warton

November 3, 2020

Problem 1

Indicate true or false.

$$\begin{aligned}
&(a) \text{ True}\\
&(b) \text{ True}\\
&(c) \text{ False}\\
&(d) \text{ True}\\
&(e) \text{ True}\\
&(f) \text{ False}
\end{aligned}$$

Problem 2

Let $T(n)$ denote the worst case running time of Sort(), and let $M(n)$ denote the running time of Merge(). Then we note the following relationship:

$$T(n) = 3 \cdot O(1) + 4 \cdot T\left(\frac{n}{4}\right) + M(n)$$

Then since we know that $M(n) = O(n)$, and $O(n) + 3 \cdot O(1) = O(n)$ we can rewrite this as

$$T(n) = 4 \cdot T\left(\frac{n}{4}\right) + O(n)$$

We take the recursive relationship and note the tree like structure giving us

$$\begin{aligned}
T(n) &= O(n) + 4 \cdot T\left(\frac{n}{4}\right)\\
&= O(n) + 4 \cdot O\left(\frac{n}{4}\right) + 16 \cdot O\left(\frac{n}{16}\right) + \cdots + 4^k \cdot T\left(\frac{n}{4^k}\right)
\end{aligned}$$

Since we know that this recursion terminates when we take $T(1)$, and since $4^{\log_4(n)} = n$, we say $k = \log_4(n)$ and that

$$O(n) + 4 \cdot O\left(\frac{n}{4}\right) + 16 \cdot O\left(\frac{n}{16}\right) + \cdots + 4^k \cdot O\left(\frac{n}{4^k}\right)$$

For this we have each level ends up being $O(n)$ time, and we have $\log_4(n)$ levels, so we must have an $\log_4(n) \cdot O(n) = O(n \log n)$ runtime.

Problem 3

To determine wether this algorithm is faster than the regular iterative one, we must check both of their running times. We know that the regular iterative one, which we will call ItMax(), will run in $\Theta(n)$ time since it will always loop through every element. Then we must compute the running time of this divide and conquer algorithm, which will be similar to that of problem 2 in some ways. We assume that $max(l_1, l_2)$ is constant time. We have the recursive relationship

$$T(n) = O(1) + 2 \cdot T\left(\frac{n}{2}\right)$$

Then if $n = 1$ we simply have $T(1) = O(1)$. We end up with the following sum

$$\begin{aligned}
T(n) &= O(1) + 2 \cdot O(1) + 4 \cdot O(1) + \cdots + 2^k \cdot O(1) \\
&= O(1 + 2 + 4 + 8 + \cdots + n) \\
&= O\left(\sum_{i=0}^{\log_2(n)} 2^i \right) \\
&= O\left(2^{\log_2(n)+1} - 1 \right) \\
&= O(2 \cdot n - 1) \\
&= O(n)
\end{aligned}$$

As it turns out, the two algoritms ItMax() and RecMax() are both in linear time, and one is not faster than the other.

Problem 4

To solve this problem in $O(\log n)$ time it is likely that we will need to use a divide and conquer approach. Let $m = \lfloor \frac{n}{2} \rfloor$. Since we have distinct numbers, if $A[m] = B[m]$, then our removed element must be some index $i \in \{m + 1, \cdots, n\}$. Otherwise, we know that $i \in \{1, \cdots, m\}$. Using this we write the following algorithm.

```
function FindMissing(A[1..n], B[1..n-1], l, r) {
    if l - r == 0:
        return l;
    else {
        result = 0;
        m = avg(l, r);
        if A[m] == B[m] {
            result = FindMissing(A, B, m+1, r);
        }
        else {
            result = FindMissing(A, B, 1, m);
        }
    }
}
```

Our base case is when we recur on an array of length 1, at which point we know we have narrowed it down to the one missing element. To quickly compute the running time, assume $avg(a, b) = O(1)$, and let $T(n)$ denote the worst case running time of FindMissing(). We have

$$\begin{aligned}
T(n) &= O(1) + T\left(\frac{n}{2}\right) \\
&= O(1) + O(1) + T\left(\frac{n}{4}\right) \\
&= O(1) + O(1) + \cdots + T\left(\frac{n}{2^{\log_2(n)}}\right) \\
&= O(1) + O(1) + \cdots + T(1) \\
&= O(1) + O(1) + \cdots + O(1) \\
&= \log_2(n) \cdot O(1) \\
&= O(\log n)
\end{aligned}$$

Thus the algorithm solves the problem in the desired running time.

Problem 5

(a) : Recursive Algorithm

Assume by the inductive hypothesis that we know $P(n - 1)$. For every string of length $n - 1$ we have two cases.

$\boxed{\text{Case 1: String Ends in 0}}$ If this is the case, then we can append only a 1 or a 2, so for each such string we have 2 valid strings of length n.

$\boxed{\text{Case 2: String Does not end in 0}}$ In this scenario, we can append a 0, 1, or 2 and have a valid string, so for each such string we have 3 valid strings of length n.
Now that this has been established, let $Q(n)$ denote the number of strings that end in 0, and $R(n)$ denote the number of strings that end in 1 or 2. We have the following recursive relationships

$$\begin{aligned} P(n) &= Q(n) + R(n) \\ Q(n) &= R(n-1) && \text{(append a 0 to a string ending in 1 or 2)} \\ R(n) &= 2 \cdot (Q(n-1) + R(n-1)) && \text{(append a 1 or a 2 to any n-1 length string)} \end{aligned}$$

So we write the algorithm as follows:

```
function NumValidStrings(n) {
    return Q(n) + R(n);
}

function Q(n) {
    if n == 1 {
        // { 0 }
        return 1;
    }
    else {
        return R(n-1);
    }
}

function R(n) {
    if n == 1 {
        // { 1, 2 }
        return 2:
    }
    else {
        return 2 * (Q(n-1) + R(n-1));
    }
}
```

(b) : Dynamic Programming

To convert this top-down recursive solution to a dynamic programming one, we will simply store $Q(n)$ and $R(n)$ in arrays and iterate up to n, then return their sum.

```
function DP_NumValidStrings(n) {
    Q,R = [] * n;
    Q[1] = 1;
    R[1] = 2;
    for i in {2,3,..,n} {
        Q[i] = R[i-1];
        R[i] = 2 * (Q[i-1] + R[i-1]);
    }
    P_n = Q[n] + R[n];
    return P_n;
}
```

(c) : Running Time

Assume that array initialization is linear time, and assume that basic arithmatic and array access by index is constant time. Then we have constant time within the for loop, and nothing that is slower than linear time, so we say that the algorithm runs in $O(n)$ time. We can write

$$T(n) = 2 \cdot O(n) + 2 \cdot O(1) + (n-1) \cdot (O(1)) + O(1) = O(n)$$

Problem 6

(a) : Algorithm

We will do a dynamic programming solution to this problem. Our first step will be to remove the complication of having black and white squares, so we loop through the matrix, and for every i, j such that $i + j$ is odd we multiply by -1. Then, we iterate from right to left, bottom to top. Computing the best score we can achieve from each tile. Let $MaxScore_{n\times n}$ be a matrix of such best scores. Denote our game board as $Board_{n\times n}$. We know that from some tile i, j we can only move down or right. Assuming that we know the best score we can achieve from either of those squares, we simply say $MaxScore[i][j] = Board[i][j] + \max\{MaxScore[i+1][j], MaxScore[i][j+1]\}$. If either of these do not exist because the index goes beyond n, just say that it's score is 0. Then since we loop from right to left, bottom to top, we guarantee that both of these tiles in question have been computed already. We write the algorithm as follows.

```
function Score(Board) {
    totalMax = -inf;
    for (i,j) in {1,2,..,n} X {1,2,..,n} {
        if i + j is odd {
            Board[i][j] *= -1;
        }
    }
    MaxScore = Board;
    for Row from n to 1 {
        for Column from n to 1 {
            if j == n and i == n {
                // Do nothing.
            }
            else if j == n {
                MaxScore[i][j] += max {0, MaxScore[i+1][j]};
            }
            else if i == n {
                MaxScore[i][j] += max {0, MaxScore[i][j+1]};
            }
            else {
                MaxScore[i][j] += max {MaxScore[i+1][j], MaxScore[i][j+1]}
            }
            totalMax = max {totalMax, MaxScore[i][j]};
        }
    }
    return totalMax;
}
```

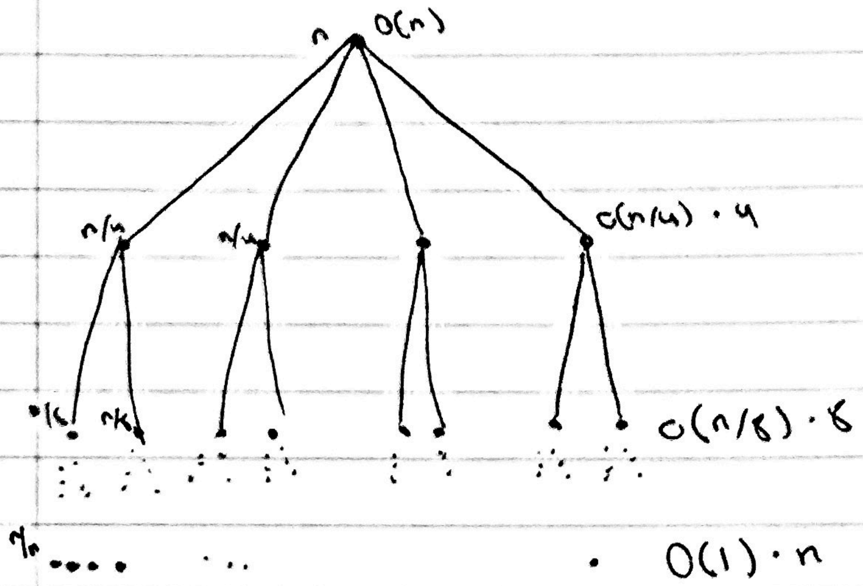
(b) : Running Time

Since we have a for loop of length n nested within another for loop of length n, it is clear that we will do no better than $O(n^2)$ running time. Let us simply compute the line by line running time of the algorithm. For the if statements, and calls to max function we say these all run in constant time.

$$T(n) = O(1) + O(n)[O(n)] + O(n)[O(n) \cdot O(1)] = O(1) + 2O(n^2) = O(n^2)$$

We can be even more precise and say that it will be $\Theta(n^2)$ since it will always perform the entirety of each double for loop.

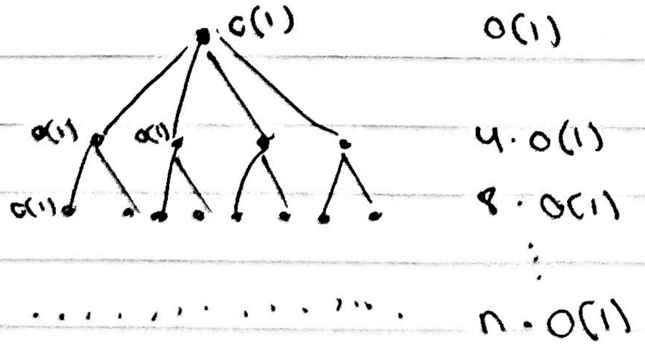
Problem 2 - Recursion Tree



- We have each layer as O(n), and $\log_4(n)$ layers. So

$$O(n) \cdot \log_4(n) = O(n \log n)$$

Problem 3 - Recursion Tree



- Count all nodes we get
$1 + 4 + 8 + \cdots + n = 4^{\log(n)+1} - 1 = 4n - 1.$
$(4n - 1) \cdot O(1) = O(n).$