HW1 Solutions

March 1, 2021

1:

(a) $f(n) = n^2 - 10n + 2$. We claim $f(n) \in O(n^2)$. For all $n \geq 1$, the following is true

$$\begin{aligned} f(n) &= n^2 - 10n + 2 \\ &< n^2 + 2 \\ &\leq n^2 + 2n^2 = 3n^2 \end{aligned}$$

Thus, $f(n) \in O(n^2)$ with $c = 3, n_0 = 1$.

(b) $f(n) = 2^{n^2}$. Assume $f(n) \in O(2^{2n})$. By assumption $\exists c, n_0 > 0$ such that the $f(n) \leq c \cdot 2^{2n}, \forall n \geq n_0$:

$$2^{n^2} \leq c2^{2n}$$

Apply $log_2(.)$ on both sides to get:

$$\begin{aligned} n^2 &\leq \log_2(c2^{2n}) \\ &= \log_2 c + 2n \end{aligned}$$

Rearranging, we get:

$$\begin{aligned} n(n-2) &\leq \log_2 c \\ n &\leq \frac{\log_2 c}{n} + 2 \end{aligned}$$

If $n > \frac{log_2 c}{n} + 2$, the inequality does not hold. This is a contradiction as our assumption states that the inequality is satisfied for all $n \geq n_0$. Thus, $f(n) \notin O(2^{2n})$

(c) $f(n) = nlog_2(n)$. We claim $f(n) \in O(nlog_{10}(n))$. For all $n \geq 1$, the following is true

$$\begin{aligned} f(n) &= n \log_2(n) \\ &= n \frac{\log_{10}(n)}{\log_{10}(2)} \end{aligned}$$

Thus, $f(n) \in O(n \log_{10}(n))$ with $c = \frac{1}{\log_{10}(2)}, n_0 = 1$.

(d) $f(n) = n \log_2(n)$. We claim $f(n) \notin O(n)$.
Assume $f(n) \in O(n)$, then there exists a constant c and n_0 such that $f(n) \leq cg(n)$ for all $n > n_0$
Then we have $n \log n \leq cn$ for $n > n_0$.
As n is positive, it is equivalent with $\log n \leq c$ for $n > n_0$
For any constant c, as n is increasing, we always have a n such that $\log_2(n) > c$
We have a contradiction, thus $f(n) \notin O(n)$.

(e) $f(n) = n2^n$. We claim $f(n) \in O(2^{2n})$.
For all $n \geq 2$, the following is true

$$\begin{aligned} f(n) &= n2^n \\ &\leq 2^{2n} = 2^n \times 2^n \end{aligned}$$

We can see both $f(n)$ and $g(n)$ have common terms 2^n, after taking the common terms from $f(n)$ and $g(n)$ the difference between $f(n)$ and $g(n)$ is $f(n)$ has n and $g(n)$ has 2^n. We know 2^n grows faster than n since the exponential function grows faster than the polynomial function(Check the textbook page 55).
Thus, $f(n) \in O(2^{2n})$ with $c = 1, n_0 = 2$.

2:

(Algo1) In the $Alg1$. Line 2 only needs constant number of operations, thus it takes constant time $c_1 = O(1)$ to terminate.

The for loop on line 3-4 iterates n times, and each iteration executes a constant number c_2 of operations. To sum them up.

$$\sum_{i=1}^{n} c_2 = c_2 * n = O(n)$$

Therefore, this for loop takes $c_2 * n$ time to complete.

The for loop on line 5-7 is nested for loop. For each $j = i$, where i is from the range from n to 1, the inner for loop iterates $j = i$ times. In another word, when $j = n$ initially, inner for loop iterates n times; when $j = n - 1$, the inner for loop iterates $n - 1$ times, so on so forth. Given that the inner for loop would take constant time c_3 in each iteration, when we sum them up, the total running time of the nested for loop is given by following equation:

$$\begin{aligned} &\sum_{j=1}^{n} \sum_{k=1}^{j} c_3 = \sum_{j=1}^{n} c_3 * j \\ &= c_3(1 + 2 + \cdots + (n-1) + n) \\ &= c_3(1+n)n/2 = O(n^2) \end{aligned}$$

Then the running time of $Alg1$ is $O(n^2)$.

(Alg2) About the $Alg2$, for each iteration, i will decrease to $i/2$, and the loop will stop when $i = 1$, since it is constant run time c for each iteration, so the total run time is $c*h$, where the h is the number of iteration. And because each time i will decrease to $i/2$, from n to 1, so there is 2^h=n, so we could get $h = log_2 n$, the total run time is $clog_2 n$ which belong to $O(log_2 n)$, The following is the math part:

$$\sum_{i=1, i=i*2}^{n} c = c * log_2 n = O(log_2 n)$$

3:

The definition of *majority element of A* is if it exists a k occurs in A strictly more than A.length /2 times, then k is the majority element of A. The idea of divide and conquer is to split the array A into two sub-arrays, let's say A_1 and A_2 of half the size of A. Then we can compute the majority element k_1 and k_2 for each sub-array A_1 and A_2. If k_1 and k_2 are equal, then they are the majority element. if not equal, we need to decide which one is the majority element or there is no majority element. For this step, we can traverse the array to count for K_1 and K_2 in linear time.
Time complexity: Each recursive to compute the majority element will performs two recursive calls on sub-problem of size $n/2$ and two linear scans of A (n operations). Therefore, the recurrence relation can be represented as:

$$T(n) = 2T(n/2) + 2n$$

You can improve this result in the scan part, in general, the recurrence relation is

$$T(n) = 2T(n/2) + O(n)$$

Then the time complexity of algorithm is $O(nlog(n))$.

Algorithm 1 FindMajority

Input: Array $A[1, \cdots, n]$
Output: Majority element of A

```
if n == 1 then
    return A[1]
m ← ⌊n/2⌋
k1 ← FindMajority(A[1, ··· , m])
k2 ← FindMajority(A[m + 1, ··· , n])
if k1 == k2 then
    return k1
/* getCount(array,k) is to compute the frequency of k in array.  */
K1count ←getCount(A[1, ··· , n],k1)
K2count ←getCount(A[1, ··· , n],k2)
if K1count > m + 1 then
    return k1
else if K2count > m + 1 then
    return k2
else
    return null
```

4:

$T(n) = 3T(n/2) + n^2$
Master Theorem $a = 3, b = 2, f(n) = n^2$ $\log_2 3 = 1.58 < 2$
Case 3 applies i..e If $f(n) = \Omega(n^{\log_b a + \epsilon})$,
then $T(n) = \theta(f(n))$.
$T(n) = f(n) = \theta(n^2)$

Proof. We see that $a = 3$, $b = 2$ and $f(n) = n^2$, $\log_2 3 = 1.585 < 2$. Let $\epsilon = 0.415$. Then

$$n^{(\log_2 3+\epsilon)} \leq n^{(1.585+\epsilon)}.$$

Therefore

$$n^2 = \Omega(n^{(\log_2 3+\epsilon)}).$$

Let $c = 3/4$. Then

$$\begin{aligned} af(n/b) &= 3(\frac{n}{2})^2 \\ &= \frac{3}{4}n^2 \\ &\leq cf(n). \end{aligned}$$

Therefore we can apply case three of the Master Theorem, and so

$$T(n) = \Theta(n^2).$$

□

5:

For brevity, all the log terms have base 2.

Recurrence Tree method

$T(n) = 2T(n/2) + nlog_2(n)$, where $T(1) = O(1)$.

Top level of the tree $= nlog(n)$
Next level of the tree $= 2 * \frac{n}{2} log(\frac{n}{2^1}) = nlog(\frac{n}{2})$
Thus the ith level of the tree $= nlog(\frac{n}{2^i})$
Height of the tree $= log(n)$, since we're dividing by 2 each time.
Last level of the tree $= n^{log_2 2} * T(1) = O(n)$
Total of the layers is $O(nlog(n))$ and with height $log(n)$, we get $O(nlog^2(n))$
$T(n) = nlog(n) + nlog(\frac{n}{2^1}) + nlog(\frac{n}{2^2}) + ... + nlog(\frac{n}{2^{lg n-1}}) + O(n)$

If we assume that n is even, then $n = 2^k$ which implies $k = log(n)$

$$\begin{aligned} T(n) &= nlog(n) + nlog(\frac{n}{2^1}) + nlog(\frac{n}{2^2}) + ... + nlog(\frac{n}{2^k}) \\ &= nk + n(k-1) + n(k-2) + ... + n(1) \\ &= n(1 + 2 + ... + k) \\ &\leq nk^2 = nlog^2(n) \end{aligned}$$

6:

Observation: The Master Theorem is applicable.

Claim 1: Professor Caesar's algorithm would be asymptotically faster than $O(n^{\log_2 3})$ if and only if

$$\log_4 a < \log_2 3. \tag{1}$$

As long as Claim 1 is true, we just need to find the largest integer value of a that satisfies inequality (1). The largest such integer value is $a = 8$, because:

$$\log_4 a < \log_2 3 \Leftrightarrow$$
$$\frac{\log_2 a}{\log_2 4} < \log_2 3 \Leftrightarrow$$
$$\frac{\log_2 a}{2} < \log_2 3 \Leftrightarrow$$
$$\log_2 a < 2\log_2 3 \Leftrightarrow$$
$$\log_2 a < \log_2 3^2 \Leftrightarrow$$
$$a < 9.$$

Proof 1. Note that we will refer to Professor Caesar's algorithm's runtime as $T(n)$. We need to prove two statements:

(i) if Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$, then $a < 9$.

(ii) If $a < 9$, then Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$.

Proof of statement (i). The following is a proof by contradiction:

Assume that $a \geq 9$. For simplicity, the following argumentation will focus on the case where $a = 9$. However, the same argumentation is also valid for any value of $a > 9$.

By the Master Theorem (first case) we get that $T(n) = \Theta(n^{\log_4 9})$. Note that $\log_4 9 = \log_2 3$, so equivalently

$$T(n) = \Theta(n^{\log_2 3}).$$

This implies that $T(n) = \Omega(n^{\log_2 3})$, but Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$, which is a contradiction.

Proof of statement (ii). The following is a constructive proof:

We will consider three cases: (a) $4 < a \leq 8$, (b) $a = 4$, (c) $1 \leq a < 4$.

(a) This is the first case of the Master Theorem, according to which,

$$T(n) = \Theta(n^{\log_4 a}), \text{ where } a \leq 8. \tag{2}$$

Since $\log_4 8 < \log_2 3$, (2) implies that Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$.

(b) This is the second case of the Master Theorem, according to which,

$$T(n) = \Theta(n \log n).$$

This implies that Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$, because $\log n = O(n^x)$ for all $x > 0$, or equivalently $n \log n = O(n^y)$ for all $y > 1$.

(c) This is the third case of the Master Theorem, according to which,

$$T(n) = \Theta(n).$$

Since $1 < \log_2 3$, this implies that Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$.

Since Professor Caesar's algorithm is asymptotically faster than $O(n^{\log_2 3})$ in each of the above three cases, statement (ii) is proven true overall.

This concludes the proof of Claim 1. Therefore, the largest integer value of a that would keep Professor Caesar's algorithm asymptotically faster than $O(n^{\log_2 3})$ is 8. □