Problemset 1

Albert Ye

January 30, 2023

1 Study Group

Zipeng Lin, 3036516523

2 Course Policies

a) Monday at 10:00 PM

b) Nothing, because homework will not be accepted late.

c) Ed

d) I have read and understood the course syllabus and policies. -Albert Ye

3 Understanding Academic Dishonesty

a) not OK

b) OK

c) not OK

d) OK

4 Recurrence Relations

a) $T(n) = 3T(n/3) + 9n$. Because the right-hand side is of the form $aT(n/b) + O(n^d)$, we can use the Master Theorem. Notice that the $9n$ term becomes $\Theta(n^1)$, and $log_3 3 = 1$. From the Master Theorem, it follows that $\boxed{T(n) = \Theta(n \log n)}$.

b) $T(n) + 4T(n/2) + n^3$. Again, we can use the Master Theorem because the right-hand side is of the form $aT(n/b) + \Theta(n^d)$. This time, $a = 4, b = 2, d = 3$, and $\log_b a = \log_2 4 = 2$. Therefore, $d > \log b_a$ and thus $\boxed{T(n) = \Theta(n^3)}$.

c) Consider that the $T\left(\frac{5n}{13}\right)$ term tends to 1 much faster than the $T\left(\frac{12n}{13}\right)$ term, so $T(n) \leq 2T\left(\frac{12n}{13}\right)$. As a result, a complexity tree is much easier to construct. Notice that this complexity tree has a height of $\log_{\frac{13}{12}}(n)$ and each level has double the amount of elements as the previous level. As a result, we have 2^i nodes in the ith row for i from 0 to $k = \log_{\frac{13}{12}}(n)$. The sum of this is equal to 2^{k+1}, or

$$\begin{aligned} 2^{\log_{\frac{13}{12}}(n)+1} &= 2 \cdot 2^{\frac{\log_2 n}{\log_2\left(\frac{13}{12}\right)}} \\ &= 2^{\log_2 n \cdot \left(\frac{\log 2}{\log \frac{13}{12}}\right)} \\ &= \left(2^{\log_2 n}\right)^{\frac{\log 2}{\log \frac{13}{12}}} \\ &= n^{\frac{\log 2}{\log \frac{13}{12}}}. \end{aligned}$$

Therefore, the complexity of $T(n)$ is bounded by $T(n) = \boxed{O(n^{\frac{\log 2}{\log \frac{13}{12}}})}$.

We can try to use a similar method to get an Ω bound. Replacing all instances of 12 with 5, we have that the complexity tree has a height of $\log_{\frac{13}{5}}(n)$ and each level has double the amount of elements as the previous level. Therefore, we have the equation

$$\begin{aligned} 2^{\log_{\frac{13}{5}}(n)+1} &= 2 \cdot 2^{\frac{\log_2 n}{\log_2\left(\frac{13}{5}\right)}} \\ &= 2^{\log_2 n \cdot \left(\frac{\log 2}{\log \frac{13}{5}}\right)} \\ &= \left(2^{\log_2 n}\right)^{\frac{\log 2}{\log \frac{13}{5}}} \\ &= n^{\frac{\log 2}{\log \frac{13}{5}}}. \end{aligned}$$

The complexity of $T(n)$ is thus also bounded by $T(n) = \boxed{\Omega(n^{\frac{\log 2}{\log \frac{13}{5}}})}$. Therefore, $T(n)$ must run in polynomial time.

After ruling out all non-polynomial answers, we can simply do guess and check. A simple plug-in of x^2 leads to a result of $\boxed{T(n) = \Theta(n^2)}$, as $T\left(\frac{13n}{13}\right) = T\left(\frac{5n}{13}\right) + \left(\frac{12n}{13}\right)$ and this is a Pythagorean triple.

Note that $\frac{\log 2}{\log \frac{13}{5}} \approx 0.8$ and $\frac{\log 2}{\log \frac{13}{12}} \approx 8.7$, so 2 is a valid exponent given the O and Ω constraints.

5 The Resistance

Let's arrange the players into an array a of length n. If we were to divide the array into $2k$ segments of equal size $\frac{n}{2k}$, note that at most k would end up failing. If we were to cut out or ignore the (at least) k segments that pass and spliced the rest of the array together, we would have an array of size $\frac{n}{2}$ with k segments.

Now, we cut each remaining segment (ignoring the ones that passed) in half to make $2k$ segments of equal size again. The size of the segments will be half, at $\frac{n}{4k}$, but the invariant that at most k will end up failing still holds.

More generally, assume we have currently performed the process i times, leaving us with k equal segments. Then, each segment is currently of size $\dfrac{n}{2^{i+1}k}$, and we have k spies distributed somehow within them. Then, we splice all the segments back into an array of size $\dfrac{n}{2^{i+1}}$. If we were to then cut each of the segments in half, we would once again have an array with $2k$ segments and k spies, meaning that once again at most k of these smaller segments would end up failing. So, as a result, we can contain all of the spies into k smaller segments of size $\dfrac{n}{2^{i+2}k}$.

Therefore, we can still continue to cut the segments that fail into smaller and smaller subsegments and splice them together until we reach the case where we have k segments of size 1. At this point, we will have reduced the list down to the list of k spies.

We originally have $2k$ subsegments of size $\frac{n}{2k}$, and we cut them all in half until they are of size 1. We would need $\log\left(\frac{n}{2k}\right)$ cuts to make that work. Moreover, there are k segments that we will end up cutting, leading to an algorithm that runs in $\Theta\left(k \log\left(\frac{n}{2k}\right)\right)$ time.

6 Werewolves

a) We simply ask all other $n-1$ people to identify the target as either a villager or a werewolf. If at least half of the players call the target a villager, then they are a villager; otherwise, they are a werewolf. This works because villagers have to tell the truth about their partner's identity, and because villagers make up at least half of the remaining players.

b) We arrange all players into an array, and keep dividing the array in half until we have segments of size ≤ 10. Now, for each of these segments, we want to find a potential villager, so we run the villager-identifying algorithm from part (a) on each player until we find a villager. If this algorithm produces a villager, then we mark it as a *potential* villager, as we do not necessarily have more villagers than werewolves for each segment, and then move onto the next segment.

After we have an initial list of candidate villagers, we now want to check each of our candidates over a larger interval. We will do this by merging two smaller intervals together, and checking each one with the villager-identifying algorithm from part (a). For each merge, if the left segment's candidate is still labelled a villager by at least half of the players in the merged segment, then we continue considering it; if not, we check if the right segment's candidate satisfies this condition. If the right segment works, then we advance it to the next level. Otherwise, we simply remove this entire merged segment from consideration. If we reach a merge where either the left or the right segment has no candidate, we can just skip that segment entirely.

We continue merging up until a point where we have two arrays of size $\frac{n}{2}$ and we want to merge them into the full array of size n. In this case, we run the same merge algorithm as described in the previous paragraph, except the candidate we promote is our answer.

Lemma. There will always be a valid villager from this algorithm.

Proof. We use the invariant that there will always be over half villagers in the whole array, and claim that the same holds for at least one of the subsegments if we cut the whole array in half.

Assume for the sake of contradiction that both segments have fewer villagers v_1, v_2 than werewolves w_1, w_2. Then, we have that $v_1 + v_2 \leq w_1 + w_2$, which is untrue because the invariant we are given states that $v_1 + v_2 > w_1 + w_2$. As a result, we need either $v_1 > w_1$ or $v_2 > w_2$.

We can apply this proof recursively to see that there is always a subsegment of length $\frac{n}{2^k}$ that satisfies our invariant for all positive integer k up until we reach $\frac{n}{2^k} \leq 10$. Therefore, at each level, we can guarantee that there exists a segment which will accurately identify a villager, and we can guarantee that that segment will also have a subsegment at the next level that will accurately identify a villager. Therefore, this algorithm is guaranteed to find a real villager. Hrn. ■

Before we analyze the entire complexity, let's look at the complexity of the base case. It's true that each individual segment runs an $\Theta(n^2)$ algorithm, but each segment is also small enough (size ≤ 10) that a $\Theta(n^2)$ algorithm still runs in constant time. There are at least $\frac{n}{10}$ subsegments in the base case, leading to the base case running in $\Theta(n)$.

For each level of merge operation, we also claim we have a $\Theta(n)$ runtime. In any arbitrary level, we have k segments of length $\frac{n}{k}$, and each merge operation takes two of the segments and merges them together into a segment of length $\frac{2n}{k}$. During each merge operation, we run the $\Theta(n)$ algorithm in part (a) twice over the segment of length $\frac{2n}{k}$, once for the candidate from each subsegment we are merging. Therefore, we have k function calls that take roughly $\frac{2n}{k}$ operations, implying that we need around $2n$ operations, for a total runtime of $\Theta(n)$ for each level of merging.

Finally, note that we have to divide the main segment $\Theta(\log n)$ times to get subsegments of length 10 or less. Therefore, we have $\Theta(n)$ operations for each of $\Theta(\log n)$ levels, giving a total run time of $\Theta(n \log n)$.