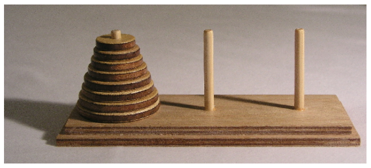
CSE102-01: Intro to Algos,[*] PSET-1 Solutions
10th January 2023

Instructions: Deadline in a week, so submit by **Wed. Jan. 17 midnight PST**. Submit to Canvas a pdf of your solutions, with your name and ucsc id number. Feel free to discuss solutions with your colleagues (other students), explore the internet, read and be inspired from the textbooks etc. But what you type should be typed by you and you only in latex. In algorithms, we appreciate brevity and accuracy: your solutions should be to the point, often one or two sentences suffice to provide all necessary details and descriptions.

Ex.1 The famous game "Towers of Hanoi" asks you to move 3 (more generally n) wooden disks from the left rod to the right rod such that a larger disk never lands on top of a smaller disk.[1] The goal is to understand how many moves it would take to perform the task. A *move* is considered an action that takes one disk from any rod and puts it in any other rod. Let $T(n)$ be the runtime of an algorithm to complete the task if there were n disks in total (play with $n = 3$ first). Provide such an algorithm whose runtime obeys: $T(n) = 2T(n-1) + 1$ (hint: recursion/induction). Then, prove that $T(n) = 2^n - 1, \forall n$. **Bonus:** As algorithm designers, the next question here is "can we do better?"



Solution Algorithm `ALG` with runtime $T(n)$: If there are n disks and 3 rods, we can first move the $n-1$ disks from their initial rod to any of the empty rods. To do so, we execute `ALG` for the $n-1$ disks that we want to move. This takes time $T(n-1)$. The next step is to move the largest disk from its initial rod to the only (legally) available rod (remember no larger disk can land on top of smaller disks). This is one move and takes constant 1 time. Then we again move the $n-1$ disks on top of the largest which takes another $T(n-1)$ time.

In total the amount spent is:

$$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1$$

which is what we wanted.

[*]PI: Vaggos Chatziafratis

[1]Check the interactive game: `https://www.mathsisfun.com/games/towerofhanoi.html`

For solving the recursion above we can use induction. We will prove that $T(n) = 2^n - 1, \forall n$.

Base case: For $n = 1$, where we have only 1 disk, 1 move suffices. Indeed, $T(1) = 2^1 - 1 = 2 - 1 = 1$.

Inductive step: Here we can assume that $T(n) = 2^n - 1$, but we have to prove that $T(n+1) = 2^{n+1} - 1$. To show this we use the fact that the recursive relationship for the runtime is: $T(n+1) = 2T(n)+1$. Now we use the inductive hypothesis for $T(n)$ which states that $T(n) = 2^n - 1$. Hence:

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

which is what we wanted to show for $T(n+1)$.

Ex.2 In algorithms, we use the Big-O notation to abstract away unnecessary details about the runtime of an algorithm.

Here is **the big-O definition (may be useful for other exercises too)**: Let $n > 0$ (usually n is the size of the input) and let $f(n)$ and $g(n)$ be two monotone and non-negative functions. We say that $f(n) = O(g(n))$ if there exists a constant c and a constant n_0 such that: $f(n) \leq cg(n), \forall n > n_0$.

The exercise asks you to use the definition to prove the following:

- Let $f(n) = 100n^2 + 10n + 1000$. Then $f(n) = O(n^2)$.
- Let $f(n) = 100n + 0.001n \log n$. Then $f(n) = O(n \log n)$.
- Let $f(n) = 50n \log n + 30n$. Then $f(n) = O(n^3)$.

Feel free to use standard properties or inequalities for the logarithm without proof.

Solution Here I will be very brief as the desired comparisons are relatively easy: For the first one, you can pick constant $c = 1110$ and $n_0 = 1$, then you will get a valid inequality for all $n > n_0$. For the second one, you can pick constant $c = 101$ and $n_0 = 2$. For the last one, you can pick constant $c = 80$ and $n_0 = 2$.

Ex.3 Imagine you had an algorithm with runtime $T(n)$ satisfying the recursion $T(n) = T(n/2) + 1$ (and $T(1) = 1$). You have probably seen the binary search algorithm that exactly obeys that recursion. Prove that $T(n) = O(\log n)$. Then, imagine another algorithm with $Q(n) = Q(n/2) + n$ (and $Q(1) = 1$). Prove that $Q(n) = O(n)$.[2]

Solution

$$T(n) = T(n/2) + 1 = (T(n/4) + 1) + 1 = \ldots = (T(n/2^i) + 1) + 1 \ldots + 1$$

where the number of 1s we need to sum up in the last expression is exactly equal to i. How many times can we do this before hitting the base case of $T(1)$?

[2]For asymptotic notation exercises, if it helps you can think that n is a power of 2.

The answer is given by solving $n/2^i = 1 \iff i = \log n$. This means that $T(n) = O(\log n)$. Moreover, you could solve the recurrence using the Master Theorem.

$$Q(n) = Q(n/2) + n = (Q(n/4) + n/2) + n = n + n/2 + n/4 + n/8 \ldots + 1$$

(here, for simplicity we assumed that n is a power of 2.)

The above sum can be rewritten as:

$$Q(n) = \sum_{i=0}^{\log n} n/2^i = n \sum_{i=0}^{\log n} 1/2^i$$

The sum is a geometric series with ratio 1/2 so:

$$\sum_{i=0}^{\log n} 1/2^i = \frac{1-\frac{1}{n}}{1-\frac{1}{2}} = 2(1-\frac{1}{n}) \leq 2$$

and hence $Q(n) \leq 2n = O(n)$.

Ex.3 Describe an $O(n \log n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Solution This is a common interview question. We search for elements a, b in the array such that $a+b = x$. The main idea is to first sort the array which takes $O(n \log n)$ time. Once the array is sorted, we fix a to be the first element of the array and then do binary search in the array for the value $x - a$. If we find such an element then we are sure that $a + (x - a) = x$ and we are done. This binary search takes $\log n$ time. We repeat this process for every element a in the array. There are n elements in total, and for each element we spent $O(\log n)$ time yielding a final $O(n \log n)$ time in total. (Question to think about: if the initial S was already sorted, could you find the two elements that sum up to x in time $O(n)$?)

Ex.4 Follow the $O(\cdot)$ definition & justify: Is $2^{n+2023} = O(2^n)$? Is $2^{2n} = O(2^n)$?

Solution Set constant $c = 2^{2024}$. Following the big-O definition we can write that:

$$2^{n+2023} \leq c2^n, \forall n \geq 1$$

and we are done (we specified the constants c, n_0 from the definition of big-O).

On the other hand, $2^{2n} = 4^n$ is not $O(2^n)$. One way to see this is by contradiction: If $4^n = O(2^n)$ then there exists constant c, n_0 :

$$4^n \leq c2^n, \forall n \geq n_0$$

This would mean that c is larger than $4^n/2^n = 2^n$ but this means c cannot be a constant (as n increases c needs to increase), which is a contradiction to the claim that c was a constant.

Ex.5 Imagine you came up with an algorithm that depends on a parameter $k \in [1, n]$ you can control. After calculations, the runtime ends up being $O(nk + \frac{n^2}{k})$. What choice of k would you make for your algorithm?

Solution We sum up two terms $nk + \frac{n^2}{k}$ and we want to minimize their sum. One way to do so (since these are positive numbers as they refer to runtimes) is to make them equal. Setting $nk = \frac{n^2}{k}$ yields $k = \sqrt{n}$ and so the final runtime is $O(n\sqrt{n})$.

Ex.7 This is from a common coding interview question: Imagine you are given a list of n integers. Assume they are distinct if that helps you. Give an algorithm that finds the smallest and the second smallest element in the list, using at most $n + \log n$ comparisons, and explain why your algorithm needs at most $n + \log n$ comparisons. (Hint: think again of the pairing process we saw in class, when trying to find the max and the min simultaneously; where is the second smallest element located? Here, thinking about tennis or chess tournaments may help you.)

Solution The main idea needed here is to create a tournament, much like tennis and chess tournaments determine their winners. We split the n numbers into $n/2$ pairs (think of n as a power of 2) and we compare them. The smaller of the two proceeds to the next round of the tournament (so the smaller number "wins" the round).

We repeat the process until there is only one number who is the winner of the tournament. By the way we defined this process we performed exactly $n-1$ comparisons and we found the smallest element. Along the way, we created the tree that is describing the pairs of numbers that got compared in each phase.

The two crucial observations are as follows: Firstly, there are at most $\log n$ numbers that got compared to the smallest element (because the tree is binary and has n leaves, so its depth must be $\log n$) and secondly, the second smallest number can only lose to the smallest number. Hence, we can backtrack in the tree and only check the list of numbers that they were ever compared with the smallest number. There are $\log n$ such numbers, and we can find the minimum among those with $\log n - 1$ extra comparisons.

The total number of comparisons is at most $n + \log n$ as desired by the exercise.