Prog 01 - COSC 3020

Thursday, September 15, 2022 9:23 PM

Beckham Carver

Question 1: Asymptotic Complexity

In the lectures, we said that logarithms with different bases don't affect the asymptotic complexity of an algorithm. Prove that $O(\log_3 n)$ is the same as $O(\log_{13} n)$. Use the mathematical definition of O– do a formal proof, not just the intuition. Start by giving the formal definition of O.

Proof: $O(\log_3 n) = O(\log_{13} n)$

-> $O(\log_3 n)$	
-> (Exists c, n_o)[$f(x) \leq c \log_3 n$](All n >n_o)	{ definition of O }
-> (Exists c, n_o)[$f(x) \leq c (\log_{13} n / 0.428..)$](All n >n_o)	{ change of bases rule }
-> (Exists c, n_o)[$f(x) \leq c * 2.335.. \log_{13} n$](All n >n_o)	{ simplification }
-> (Exists c, n_o)[$f(x) \leq c \log_{13} n$](All n >n_o)	{ combine c and 2.335 }***
-> $O(\log_{13} n)$	{ definition of O }

here we combine the constant 'c' with the constant 2.335, because in terms of asymptotic complexity a larger constant is still a constant. ***It's property as a constant is unchanged.

Proof: $O(\log_{13} n) = O(\log_3 n)$

-> $O(\log_{13} n)$	
-> (Exists c, n_o)[$f(x) \leq c \log_{13} n$](All n >n_o)	{ definition of O }
-> (Exists c, n_o)[$f(x) \leq c (\log_3 n / 2.335..)$](All n >n_o)	{ change of bases rule }
-> (Exists c, n_o)[$f(x) \leq c * 0.428.. \log_3 n$](All n >n_o)	{ simplification }
-> (Exists c, n_o)[$f(x) \leq c \log_{13} n$](All n >n_o)	{ combine c and 0.428 }***
-> $O(\log_3 n)$	{ definition of O }

here we combine the constant 'c' with the constant 0.183, because in terms of asymptotic complexity a smaller constant is still a constant. ***It's property as a constant is unchanged.

Because the equality is proven to work in both directions, $O(\log_{13} n) = O(\log_3 n)$.

Question 2: Runtime Analysis

Analyze the running time of the following recursive procedure as a function of n and find a tight big-O bound on the runtime for the function. You may assume that each operation takes unit time. You do not need to provide a formal proof, but you should show your work: at a minimum, show the recurrence relation you derive for the runtime of the code, and then how you solved the recurrence relation.

Code	**Iteration Cost**	**Big O**
function mystery(n) {		
if(n <= 1)	c	1
return;		
else {		
mystery(n / 3);	T(n/3)	$\log_3 n$
var count = 0;		
mystery(n / 3);	T(n/3)	$\log_3 n$
for(var i = 0; i < n*n; i++) {	n^2 * (inner)	$O(n^2)$ * (inner)
for(var k = 0; k < n*n; k++) {	n^2	$O(n^2)$
count = count + 1;		
}		
}		
mystery(n / 3);	T(n/3)	$\log_3 n$
}		
}		

Recurrence relation:

T(n) = 1	if n <= 1
T(n) = 3T(n/3) + $O(n^4)$	if n > 1

The first iteration of the function will always run at $O(n^4)$, all subsequent iterations within the recursion tree will be asymptotically smaller than the first. Notable 3 * $(n/3)^4$ is smaller than n^4 in all but the base case 1. This means that our recurrence relation evaluates to O(less than n^4) + $O(n^4)$ which is **equal to $O(n^4)$**.

Formally proving this using geometric series is painful and cruel and honestly speaking should be considered a crime. It essentially breaks down to the worded proof above where the series equates to k = log3(n). The 3T(n/3) side is negated once it is solved because it is smaller than $O(n^4)$. Case 3 of the masters theorem confirms this as well.

Question 3: Sorting - Insertion Sort

Sort the list 1,9,0,1,3,8 using insertion sort, ascending. Show the list after each outer loop. Do this manually, i.e. step through the algorithm yourself without a computer

i	Array	Logic Summary
DNE	[4, **3**, 1, 0, -1, 8]	4 > 3 : insert '3'
1	[3, 4, **1**, 0, -1, 8]	4 > 1 : insert '1'
2	[1, 3, 4, **0**, -1, 8]	4 > 0 : insert '0'
3	[0, 1, 3, 4, **-1**, 8]	4 > -1 : insert '-1'
4	[-1, 0, 1, 3, 4, **8**]	4 < 8 : no change
5	[-1, 0, 1, 3, 4, 8]	i = 5 : array sorted

Question 4: Sorting - Merge Sort

(See attached code for part 1 of this question)
Analyze the time complexity of your implementation and give a Θ-bound for its worst-case runtime:

The worst case runtime for my implementation is Θ(n log n), the merge() portion of the function will take **at most** n-time for each subarray where n is the size of each subarray. By iterating through all subarrays the total size of n given to merge is the size of the whole array. This merge() function is called log n times from msort(). This is because the loops breaking up the array for merge() are increasing by a factor of 2 up to the size of the array.

Question 5: Sorting - Quick Sort

Using the probabilities for picking a pivot in a particular part of the array (in the same way as we did on slide 34), argue whether this method is more or less (or equally) likely to pick a good pivot compared to simply choosing the first element. Assume that all permutations are equally likely, i.e. the input array is ordered randomly. Your answer must derive probabilities for choosing a good pivot and quantitatively reason with them to get the answer

At a trivial level, median of three always ensures that we make not the worst pick out of three options, which means at a minimum it is at least the 2rd worst pick of all possible. Whereas with always picking the first element, our worst case is that is the worst possible pick. Median of three also ensures that we account for sorted/reverse sorted arrays.

Median of three is more likely to pick a good pivot than choosing the first element. This is because by choosing a median of three we have guaranteed that our chosen element is statistically more likely to be a truer median than a complete random pick.

- When choosing an arbitrary element it is most likely to be greater than half of the other elements, however being the most likely of an case does not make it a likely case.

    - A random number chosen from a bag with #1-10 in it, is equally likely to be any number. There is a 10% chance we will pick a 5.
    - Treating 4,5,6 as 'good' picks, these make up 30% of the bag, meaning we have a 70% chance to make a bad pick and we stick with it.
- This seems like okay odds, however lets compare to median of three with a similar experiment.
    - With median of three, we select three items from this bag. Our first pick has a 70% chance of being **bad**, if it is a poor pick, then our next pick has a 66% chance of being **bad**. If it is also a poor pick, then our last draw has a 62.5% chance of being **bad**.
    - Notably bad picks will always be outliers, larger or smaller than average, which will inform our choice of the picks. Meaning we always make the best decision of our three choices.
    - Averaging these three picks and knowing we make the best choice of the three, we have a 66.16% chance of making a bad pick, which is a 3% improvement over the 70% chance.
    - Furthermore, if a good pick is ever made, it will always stand out amongst the outliers.