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 $O(\log_2 n)$is the same as $O(\log_{10} n)$. Use the mathematical definition of O– do a formal proof, not just the intuition. Start by giving the formal definition of O.

$O(\log_2 n)$ is equivalent to $O(\log_{10} n)$ if both $\log_{10} n \in O(\log_2 n)$ and $\log_2 n \in O(\log_{10} n)$

Proof: $\log_{10} n \in O(\log_2 n)$

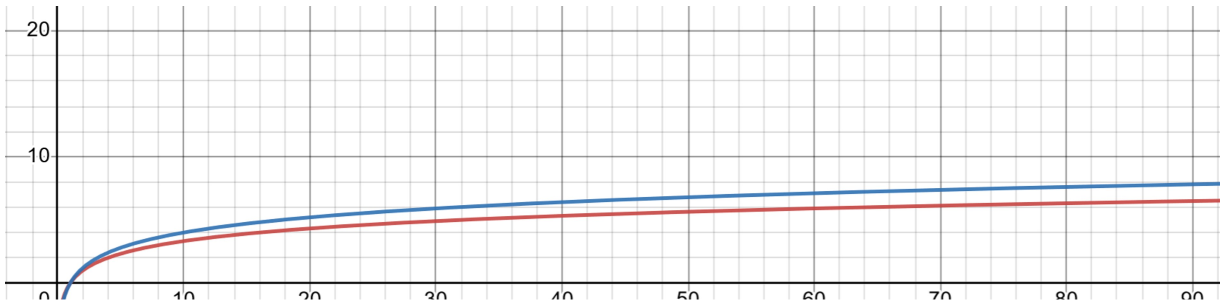
-> (All n)(Exists c)(Exists n_0) [$n > n_0$ *implies* $\log_{10} n \le c \log_2 n$]	{ definition of O }
-> (Exists c)(Exists n_0) [$n > n_0$ *implies* $\log_{10} n \le c \log_2 n$]	{ remove (All n) for n }
-> (Exists n_0) [$n > n_0$ *implies* $\log_{10} n \le \log_2 n$]	{ remove (Exists c) for 1 }
-> $n > 1$ *implies* $\log_{10} n \le \log_2 n$	{ remove (Exists n_0) for 1 }
-> true *implies* $\log_{10} n \le \log_2 n$	{ property assumed true }
-> true *implies* true	{ properties of logarithms }
-> true	{ boolean logic }

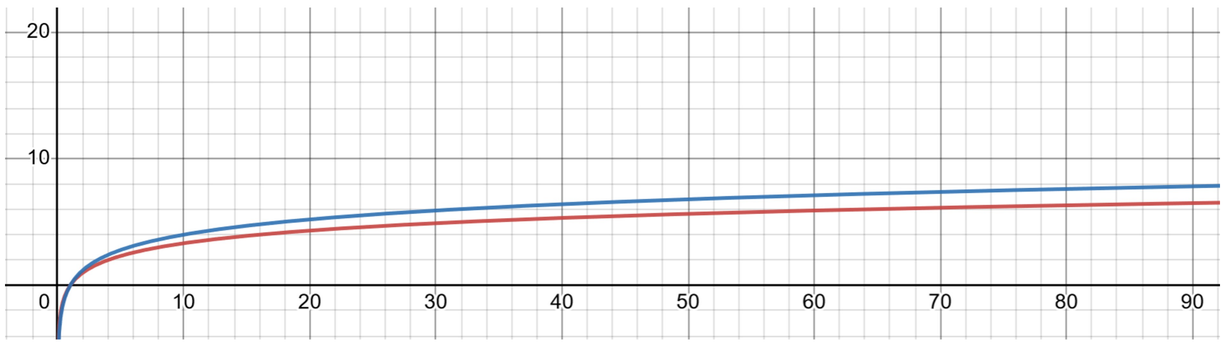
The above proof is trivial because $\log_2 n$ is naturally larger than $\log_{10} n$.

Proof: $\log_2 n \in O(\log_{10} n)$

-> (All n)(Exists c)(Exists n_0) [$n > n_0$ *implies* $\log_2 n \le c \log_{10} n$]	{ definition of O }
-> (Exists c)(Exists n_0) [$n > n_0$ *implies* $\log_{10} n \le c \log_2 n$]	{ remove (All n) for n }
-> (Exists n_0) [$n > n_0$ *implies* $\log_{10} n \le 4 \log_2 n$]	{ remove (Exists c) for 4 }
-> $n > 1$ *implies* $\log_{10} n \le \log_2 n$	{ remove (Exists n_0) for 1 }
-> true *implies* $\log_{10} n \le \log_2 n$	{ property assumed true }
-> true *implies* true	{ properties of logarithms }
-> true	{ boolean logic }

The above proof can also be visualized with the graph below, where the blue line represents $y = 4 \log_{10} x$ and the red represents $y = \log_2 x$





Because both functions belong to the *O* of the other functions, they are asymptotically equivalent.

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.

```
function mystery(n) {
    if(n <= 1)
        return;
    else {
        mystery(n / 2);
        var count = 0;
        for(var i = 0; i < n*n; i++) {
            for(var j = 0; j < n*n; j++) {
                count = count + 1;
            }
        }
        mystery(n / 2);
    }
}
```

We know that a recursive function that calls itself with (n/2) items belongs to O(n log n). There are two mystery(n/2) calls, meaning that the mystery(n) function itself is called **2 * n log(n) times.** Following each of these calls we can derive the runtime below:

The base case is a single operation, called once, which is constant.
1 operation

The else case is called 2 * n log(n) times, and contains a initializing operation each time.
2 * n log(n) operations

The else case is called 2 * n log(n) times, it contains two loops. The outermost loop will be called 2 * n log(n) times and initialize each time.
2 * n log(n) operations

The else case is called 2 * n log(n) times, with the outermost loop initialized it will run the inner loop for $(2 * n \log(n))^2$ times. Each time the inner loop is called it is initialized.
$(2 * n \log(n))^2$ operations

The inner loops is called $(2 * n \log(n))^2$ times. Each time it is called it performs a single operation $[(2 * n \log(n))^2]^2$ times.
$[(2 * n \log(n))^2]^2$ operations

Adding these together we get the total runtime of:
$[(2 * n \log(n))^2]^2 + (2 * n \log(n))^2 + 2 * n \log(n) + 2 * n \log(n) + 1$

Which *(somewhat)* simplifies to:
$(2 * n \log(n))^4 + (2 * n \log(n))^2 + 2(2 * n \log(n)) + 1$

In terms of a tight big-O the most relevant term is n^4 as this will quickly outpace all other terms in the function.

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	[1, 9, 0, 1, 3, 8]	1 < 9 : no change
1	[1, 9, 0, 1, 3, 8]	9 > 0 : insert '0'
2	[0, 1, 9, 1, 3, 8]	9 > 1 : insert '1'
3	[0, 1, 1, 9, 3, 8]	9 > 3 : insert '3'
4	[0, 1, 1, 3, 9, 8]	9 > 8 : insert '8'
5	[0, 1, 1, 3, 8, 9]	--

Question 4: Sorting - Merge Sort

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

The first portion of merge sort is the same as discussed in class. If the array has 0 or 1 elements, then it is sorted.
1 operation

Split the array in two equal sized halves
1 operation

Then sort each half recursively using merge sort. This recursive call will ALWAYS be done **log(n) times** as shown in the lecture slides. During each of these log(n) calls the following operations are made:

> The selected array will be iterated through for comparison, as our start approaches our sort-point and our second array start approaches the end of the array selection. During this iteration, x[a] is compared to x[b]
> **1/2 * n * log(n) * 1 operations**
>
> Assuming the worst case, an element from the second array is found to be smaller than the first array, all elements between them are shifted. At most this is half of the selected portion of the array. This can happen 1/2 * n times.
> **1/2 * n * log(n) operations**

Adding these together we get:
Log(n) * [(1/2 * n * log(n)) + (1/2 * n * log(n))]

Which simplifies to:
Log(n) * n * log(n)

Our worst case Θ-bound of in-place merge sort is log(n)*n*log(n)

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

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.