Name: Georges Hatem

CS590 Homework 1 Analyzing Algorithms Reinforcement Exercises

Due Date: January 30, 2022

<u>Problem 1.6.7:</u>

The following list of functions below were ordered by the big-Oh notation from increasing to decreasing. Section 1.2 Table 1.2.4 was used to order the following functions as well as reasoning and when in doubt between 2 functions, plug in large multiple numbers to check which functions is higher

<u>Ordering the following list of functions by the big-Oh notation from increasing to decreasing:</u>

1) 2^{2^n}
2) 4^(n)
3) 2^(n)
4) n^(3)
5) (n^2)*(log(n))
6) 4^log(n)
7) 4n^(3/2)
8) 2*n*(log(n))^(2)
9) 6*n*log(n)

10) $n\log_4(n)$
11) 5n
12) $2^{\log(n)}$
13) 3*n^(0.5)
14) $\sqrt{n}$
15) n^(0.01)
16) (log(n))^(2)
17) $\sqrt{\log(n)}$
18) log(log(n))
19) 2^(100)
20) 1/n

<u>Grouping Together functions that are big-Theta of one another:</u>

1) 6*n*log(n) and n*$\log_4$(n) are big-Theta of one another (Reasoning below)
2) 2^(log(n)) and 5n are big-Theta of one another (Reasoning below)
3) $3 * (n^{0.5}) and \sqrt{n}$ are big-Theta of one another (Reasoning below)

<u>Reasoning for $3 * (n^{0.5}) and \sqrt{n}$ are big-Theta of one another:</u>

$$n^{0.5} = \sqrt{n}$$

This means the following:

$$3 * (n^{0.5}) = 3 * \sqrt{n}$$

$3 * \sqrt{n}\ and\ \sqrt{n}$ are big-Theta of one another because the only difference is that one of them is multiplied by a contact, which is 3 in this case.

<u>Reasoning for 2^(log(n)) and 5n are big-Theta of one another:</u>

Let's consider $n = 2^x$, then we have the following:

$$2^{\log(n)} = 2^{\log(2^x)} = 2^x\ and\ 5n = 5 * (2^x)$$

Both functions are big-Theta of one another because the only difference is that one is multiplied by a constant, which is 5 in this case.

<u>Reasoning for 6nlog(n) and nlog$_4$(n) are big-Theta of one another:</u>

Let's consider as the previous example that

$$n = 2^x$$

This means the following:

$$\begin{aligned} 6 * n * \log(n) &= 6 * (2^x) * \log(2^x) \\ &= 6 * x * (2^x) \end{aligned}$$

$$n * \log_4(n) = (2^x) * (\log_4 2^x)$$

Now, let's divide x by 2 and multiply by 2 like nothing is done just so that we can square the 2 to get a 4:

$$n * \log_4(n)$$
$$= (2^x) * (\log_4 \left(2^{2*\frac{x}{2}}\right)) = (2^x) * (\log_4 4^\wedge(\frac{x}{2}))$$
$$n * \log_4(n) = (2^x) * \left(\frac{x}{2}\right) = \left(\frac{1}{2}\right) * (x) * (2^x)$$

So, 6nlog(n) is as follows:

$$6 * n * \log(n) = 6 * (2^x) * \log(2^x)$$
$$= 6 * x * (2^x)$$

And $n\log_4(n)$ is as follows:

$$n \log_4(n) = \left(\frac{1}{2}\right) * (x) * (2^x)$$

So, from this, we can deduce that both functions are big-Theta of one another because the only difference is that both functions are multiplied by a different constant.

Problem 1.6.9:

Looking at Figure 1.3.2 in Section 1.3, I will start analyzing the primitive operations to determine the runtime of the Algorithm given in terms of n.

Figure 1.3.2: Algorithm

Algorithm arrayFind(x, A):

Input: An element x and an n-element array, A.

Output: The index i such that $x = A[i]$ or -1 if no element of A is equal to x.

$i \leftarrow 0$
while $i < n$ **do**
    if $x = A[i]$ **then**
        return i
    else
        $i \leftarrow i + 1$
return -1

Feedback?

One of the first primitive operation that come to mind is assigning i = 0. This would be counted as 1 primitive operation.

Another primitive operation that come to mind is the comparison between i and n. This primitive operation will run n+1 times because i starts at 0. n is the length of the array. The instructions inside of the while loop will run n times. However, the comparison runs 1 time more, which means n+1, because it would also get tested for n. So, the primitive operation for the comparison between i and n is n + 1.

Another primitive operation that comes to mind is checking whether x is equal to A[i]. This primitive operation will run n times in the worst case scenario. In the worse-case scenario, a value of -1 will be returned and no x will be matched with any row in the n*n array. So, the worst-case scenario is when we check whether x is equal to A[i] for all the rows in the n*n array. So, the primitive operation of checking whether x is equal to A[i] is n.

return i will never make it in the worst-case scenario. In fact, to get the worst-case scenario, the algorithm should skip over return i.

Another primitive operation that comes to mind is the 2 primitive operations produced by i = i + 1. First, we add i to 1, and this is one primitive operation and then we assign i with the value of i + 1. These 2 primitive operations will always be running (n times). So, the primitive operation proceduced by i = i +1 is (2)*(n).

Another primitive operation that comes to mind is return -1. This return -1 occurs in the worst-case scenario. It occurs after all values in a single row are checked and x does not match any value in the arrays row.

After Jotting all of them above, let’s calculate the worst-case running time of arrayFind in terms of n,

$$(1) + (n + 1) + (3 * n) + 1 = 4n + 3 = O(n)$$

Now, it is mentioned in the Exercise that find2D irritates over the rows of A and calls the Algorithm arrayFind on each one until x is found or it is has searched all rows of A.

This means that find2D will repeat the same thing that arrayFind did for each row. Since we have n rows, then the worst-case runtime of find2D in terms of n is O(n^2).

Therefore, in conclusion the worst-case runtime of find2D in terms of n is O(n^2).

<u>Is this a linear Algorithm? Why or why not?</u>

Since the worst-case runtime of find2D in terms of n is O(n^2), the Algorithm is not considered linear. An Algorithm is said to have linear time if its worst-case runtime is O(n).

Since the worst-case runtime for find2D is O(n^2), then it is not a linear time Algorithm.

<u>Problem 1.6.22:</u>

As stated in Section 1.2 after Example 1.2.9, f(n) is o(g(n)) if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

provided this limit exists

So, let's apply this into practice to show that

$$n\ is\ o(\mathrm{n} * \log(n))$$

By looking at the limit formula above and at the problem question that we want to solve, we can easily determine the following:

$$f(n) = n\ and\ g(n) = n * \log{(n)}$$

So, let’s solve for the limit and check if the limit exist and we get 0:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{(n)}{(n * \log(n))} = \lim_{n\to\infty}\frac{1}{\log(n)}$$

log(n) increases when n increases and reaches very high values when n is very high. This means that log(n) at plus infinity is plus infinity. 1/infinity is 0:

$$\lim_{n\to\infty}\frac{1}{\log(n)} = 0$$

Therefore,

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$$

This shows that n is o(nlog(n))