Homework 2 – Answers to Questions

Jade Pearl

CMSC 451-7380

Professor Dennis Didulo

**Question 1**:

Given the two functions:

* *f*(*n*) = 6*n*3 + 4*n*2 + 2
* *g*(*n*) = 5*n*2 + 9

The following can be proved/disproved using L’Hopital’s Rule and limits:

* *f* 🡪 this was proven
* *g*  🡪 this was disproven (SEE THE WORK BELOW)

For *f* ɛ Ω(*g*), evaluate the limit if possible:

Use L’Hopital’s Rule to reevaluate the limit since it was indeterminate.

which is still indeterminate.

Reapply L’Hopital’s Rule until indeterminate is not reached:

because 36\*∞+ 8 = ∞ and ∞/10 = ∞

We now need to take into account the Limit rule for Ω-notation: ≠ 0 (either a strictly positive constant or infinity), then f ɛ Ω(g). Using limit rules, the limit from n 🡪 ∞ for f(n)/g(n) is ∞. So, we have proven that *f* ɛ Ω(*g*).

Do the same thing for *g* ɛ Ɵ(*f*). Evaluate the limit if possible (remember to use g(n)/f(n) instead of f(n)/g(n) since we are using the Ɵ rule for f(n) not g(n):

which is indeterminate.

Apply L’Hopital’s rule:

because when dividing by ∞ in limits, the limit approaches 0 therefore is 0.

The limit rule for Ɵ-notation in terms of *g* is: where *c* is a positive constant (>0) and not infinity, then *g* ɛ Ɵ(*f*). Using limit rules, the limit from n🡪infinity for g(n)/f(n) is 0 which is not a constant greater than 0. This disproves that *g* ɛ Ɵ(*f*).

**Question 2**:

The following functions are ranked in asymptotic from lowest to highest:

* log­4 *n*
* 2log2 *n*
* 3√𝑛
* 4*n* + 10
* 3*n* log2 *n*
* *n*2 + 6*n*
* 2*n*2 + 10*n* + 5
* 2*n*3 + *n*2 + 6
* 2*n*
* 4*n*

**Question 3**:

The recursion tree for the given function has each node that represents each time the sumSquares recursive function is called and the “first” and “last” index values passed:

T(n)



T(n/2)



T(n/2)



T(n/4)



T(n/8)



T(n/8)

The formula for the number of nodes in the recursion tree with an array of size *n* would be 2n-1 because with each new recursive call after the first depth of the tree that isn’t the root node, the nodes split off into two children and only one of those children have children where the other child does not (leaf node). To test this formula, we plug in *n* = 12.

2(12) -1 = 24 – 1 = 23, which based on the recursion tree drawn above, there are in fact 23 nodes counting the root node. So this formula is correct.

Let’s also test base case where n = 1.

2(1) – 1 = 2 – 1 = 1 which holds true because if the array is size 1, the program starts with first == last and so there will be only one node and no further recursion.

To find the Big-Θ for execution time, we know that the formula 2n – 1 for the number of nodes holds true for any number of *n* (any array length), so we can conclude that the Big-Θ for execution time sums up to f(n) = Θ(*n*) for f ɛ Θ(*n*).

The height of the tree can be determined how many times you can divide *n* by 2 until 1 is reached. So the formula for the height of the tree is log2 n. In the case where n = 12, the height is 4 (does not include the root node).

The Big-Θ for memory is determined by the formula for the height of the tree. Since the formula for height is log­­2 *n*, then the Big-Θ for memory = Θ(log n).

**Question 4**:

The initial condition, the base case is where n, the size of the array, is equal to 1. If n = 1, the function returns array[first] \* array[first] and there are no more calls to sumSquares. T(1) = 1 is the base case.

The recurrence equation is represented by the work done for recursive calls on two subproblems on size n/2 + 1 where the 1 represents the constant work for dividing the problem and combining the results. So, the recurrence equation is T(*n*) = 2T(*n*/2) + 1.

The critical exponent is found from the branching (b) and cutting (c) factors of the recurrence equation 🡪 T(n) = bT(n/c) + 1. The formula for the critical exponent is:

E = log(b)/log(c) = log(2)/log(2) = 1. The critical exponent is 1.

The row sums of the tree increase geometrically since the number of leaves is nE. When n = 12 and the critical exponent is used, the number of leaves is correct as shown in the recursion tree: 121 = 12.

This shows that case 1 of the Little Master Theorem applies where T(n) ɛ Θ(nE) = Θ(n).

To check the optimality of the algorithm, we must consider the time complexity and the logic of the sumSquares. The time complexity of Θ(n) indicates a linear time complexity which shows direct proportionality to the size of the array in this case. For computing the sum of squares of elements in an array, such time complexity is optimal because you must go through the elements of the array a minimum of one time before getting the results. If it were somehow possible to take the sum of squared elements in the array without inspecting the elements (which is impossible) at least once, then this time complexity would not be optimal. Since such a thing is not possible, a linear time complexity of Θ(*n*) is the most optimal for such a task.

**Resources**

*L’Hopital’s Rule*. (2021, February 22). Calcworkshop. <https://calcworkshop.com/derivatives/lhopitals-rule/#:~:text=Observe%20that%20we%20had%20to>

Downey, A. (n.d.). *13.1: Order of Growth* [Review of *13.1: Order of Growth*]. LibreTexts Engineering. <https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Think_Python_-_How_to_Think_Like_a_Computer_Scientist_(Downey)/13%3A_Appendix_B-_Analysis_of_Algorithms/13.01%3A_Order_of_Growth#:~:text=An%20order%20of%20growth%20is,set%20grows%20linearly%20with%20n>.