Questions and/or Exercises to work out and turn in:

Grading Guidelines (See Appendix About Getting Full Credit):

* A right answer will get full credit when:

1. It is right (worth 25%)
2. It is right **AND** neatly presented making it easy and pleasant to read. (worth an **extra** 15%)
3. There is an **obvious and clear link[[1]](#footnote-1)** between 1) the information provided in the exercise and in class and 2) the final answer. A clear link is built by properly writing, justifying, and documenting an answer (worth an **extra** 60%).
4. Calculation mistakes will be minimally penalized (2 to 5% of full credit) while errors on units will be more heavily penalized.

You are welcome/encouraged to discuss exercises with other students or the instructor. But, ultimately, **personal** writing is expected.

* USE THIS FILE AS THE STARTING DOCUMENT YOU WILL TURN IN. **DO NOT DELETE ANYTHING FROM THIS FILE:** JUST **INSERT** EACH ANSWER **RIGHT AFTER ITS QUESTION/PROMPT**.
* IF USING HAND WRITING (STRONGLY DISCOURAGED), **USE THIS FILE** BY CREATING SUFFICIENT SPACE AND WRITE IN YOUR ANSWERS.
* FAILING TO FOLLOW TURN IN DIRECTIONS /GUIDELINES WILL COST **A 30% PENALTY.**

Objectives of this assignment:

* to use and manipulate the concepts presented in this module
* to use and manipulate the definitions of O(g(n)), Ω(g(n)), and Θ(g(n))
* to get familiar with the “order” of usual functions: polynomials, square root, logarithms, exponentiatl...

What you need to do:

Answer the questions and/or solve the exercises described below.

Exercise 1 (25 points) (See Appendix...)

Consider two algorithms A1 and A2 that have the running times T1(n) and T2(n), respectively.

and

1. (24 points) **Use the definition** **of O()** used in this course (textbook) to show that T2(n) O(T1(n)). **Hint**: See Slide 28 or Textbook Pages 44-49. You will have to find/exhibit/provide the numbers and

One of the first observations when looking at T1(n) and T2(n) is how close they resemble each other, with exception to the leading coefficients from each formula. T1(n) is exactly 1/50th in scale for any value of n for T2(n). This is due to dividing the leading coefficients from each formula which is 200/4. This gives a value of 50 with this being the scaled-up factor of T1(n). We need T2(n) to be less than or equal to the value we choose for c times T1(n). For c = 1 and n0 = 1, it is clear that T2(n) satisfies both and is always being greater than 0 and then always being less than or equal to T1(n) for all values of n that are greater than or equal to n0. Therefore, it can also conclusively demonstrate that T2(n) is in O(T1(n)) by satisfying the necessary conditions for the constants c and n0, proving the upper bound relationship.

1. (1 points) Which algorithm should you use? A1 or A2?

This first exercise is meant to be a very simple form of proving this relationship. Since the only thing we are worried about is the coefficients to a certain extent, since these are the only values in either formula that differ. So since A2 will always have a better run time for any positive value over A1 it would always be the better and more efficient formula. Again this is even with the base value of c = 1 and n0 = 1.

Exercise 2 (20 points) (See Appendix...)

Consider two algorithms A1 and A2 that have the running times T1(n) and T2(n), respectively.

and are as defined in Exercise 1

1. (15 points) **Use the definition of Ω()** used in this course (textbook) to show that T2(n) Ω(T1(n)). **Hint**: See Slide 28 or Textbook Pages 44-49. You will have to find/exhibit the numbers and

By examining the growth rates of the two algorithms, A1 and A2, with their respective running times, we find that A2's running time maintains a consistent proportionality to A1's running time, which allows us to demonstrate that A2's running time is in the omega of A1's running time. Choosing a constant c as 1/50 effectively scales A1's running time to a level where it's always less than or equal to A2's running time for all n greater than or equal to 1. This comparison directly adheres to the omega notation's definition, showing that for all sufficiently large n, A2's running time is at least a constant factor of A1's running time, thereby establishing the required to prove that T2(n) is in the omega of T1(n).

1. (5 points) **Show** that T1(n) using the most compelling and **concise** arguments. **Hint**: use what you already established in this exercise and the previous exercise.

The relationship between the first and the second algorithms' running times can be understood through their growth rates, which are dictated by the same highest order terms, albeit with different coefficients. Since we've already established that the second algorithm's running time forms a lower bound to the first, and given their identical order of growth, it's clear that the first algorithm's running time also forms an upper bound to the second. This symmetry in their growth rates demonstrates that the running times of the two algorithms are tightly bound to each other, indicating a Θ relationship.

Exercise 3 (20 points)

List all the functions below from the lowest to the highest order (in terms of growth). Functions with similar growth must be grouped between brackets ([]). Brief justifications are ok.

5\*en+1 100n 10\*7n n2lg(n) (nln(n))2 lg(n) n4

en n2 7n-1 lg(lg(n)) n (lg(n))2 n! n – n4 + 7n6

One thing to remember when these have been classified below is if there is a higher order of growth included in the function, like if there is both a polynomial and an exponential in the same function, this function would still show exponential growth overall, as it would still dominate the polynomial.

**Logarithmic Growth –** These three functions grow in similar growth patterns based on their logarithmic function. These functions will grow the slowest out of all the listed functions. They are also listed in the order from lowest to fastest within logarithmic growth patterns.

**( lg(lg(n)), lg(n), (lg(n))2 )**

**Linear Growth –** This is technically the only listed pure linear growth pattern listed and will grow faster than logarithmic functions above but slower than the polynomial functions listed below it.

**( 100n )**

**Polynomial Growth –** These functions will grow faster than both logarithmic and linear functions but not faster than exponential or factorial functions.

**( n2lg(n), (nln(n))2, , n, n – n4 + 7n6 )**

**Exponential Growth –** These functions will grow faster than most all functions except factorial growth of related functions. When looking at this type of growth it is important to pay attention to the power at which the exponent is.

( 5\*en+1, 10\*7n , n4, n2, en)

**Factorial Growth –** These are considered some of the most fastest growing functions over all the other types. As the number for n grows so does how much growth can be shown.

( n!)

Exercise 4 (35 points) (See Appendix...)

Consider the algorithm *getIndexMaximum(A,k)* that takes a sequence A as an input and returns the index of the largest number (maximum) in the range [k-A.length] in Sequence A.

For example, let A ={100, 2, 14, 5, 22, 7}. *getIndexMaximum(A,3) will return 5 because 5 is the index of the element 22 and 22 is the ;argest number in A in the range [3-6].*

We assume that *getIndexMaximum(A,k)* performs 2(n-k)+2 comparisons where n = A.length (it seems that *getIndexMaximum(A,k)* is not efficient).

Consider the following sorting algorithm that sorts a sequence A in decreasing order:

Sort-Array(A)

0: n = A.length

1: for i = 1 to n

2: IndexMax = getIndexMaximum(A,i)

// swap A[i] and A[IndexMax]

3: buffer = A[IndexMax]

4: A[IndexMax] = A[i]

5: A[i] = buffer

The objective is to find the total number comparisons performed by the above algorithm Sort-Array:

* 1. (1 point) How many comparisons in total are performed by the “*for loop*” statement in Line 1 during the execution of the algorithm?

The for loop will iterate from 1 to n where n is the length of the array A. For each iteration I, the function getIndexMaximum(A,i) is called, which performs 2(n-i) + 2. When implemented and simplified the total number of comparisons the for loop will be n(n + 1). Directly related to just the line 1 there will be n comparisons though.

* 1. (20 points) Let us call cj the number of comparisons performed by *getIndexMaximum(A,i)* (Line 2). Fill in this table (**Justify** how you determine cj):

|  |  |
| --- | --- |
| i | ci |
| 1 | when i = 1, getIndexMaximum(A,1) performs 2(n-1)+2... comparisons. |
| 2 | when i = 2, getIndexMaximum(A,2) performs 2(n-2)+2... comparisons. |
| 3 | when i = 3, getIndexMaximum(A,3) performs 2(n-3)+2... comparisons. |
| k | when i = k, getIndexMaximum(A,k) performs 2(n-k)+2... comparisons. |
| n | when i = n, getIndexMaximum(A,n) performs 2... comparisons. |

This table reflects the decreasing number of comparisons as i increases, since fewer elements are considered for finding the maximum as i moves from 1 towards n.

When i = 1: This accounts for comparing each element in the array except the first one with a potential maximum and then comparing the maximum found with the first element to determine if a swap is needed.

When i = 2: This time, the comparisons are for finding the maximum in the array starting from the second element onwards.

When i = 3: reflecting the comparisons to find the maximum from the third element to the end of the array.

When i = k: this formula adjusts dynamically based on the value of k, representing the comparisons from the kth element to the last element.

When i = n: At this point, since i equals the length of the array, the function effectively checks if the last element is greater than itself, resulting in a minimal number of comparisons, just 2, as there are no elements after the last one to compare with.

* 1. (10 points) Based on b, express the **total** number of comparisons performed by *getIndexMaximum(A,j)* in Line 2 during the full execution of the algorithm.

The for loop will iterate from 1 to n where n is the length of the array A. For each iteration I, the function getIndexMaximum(A,i) is called, which performs 2(n-i) + 2. When implemented and simplified the total number of comparisons the for loop will be n(n + 1).

* 1. (1 point) Express the function fc(n) that represents the **overall total** number of comparisons performed by the statements in Lines 2 and 1 during the full execution of the algorithm.

Fc(n) = n(n+1) would be the overall total number of comparisons.

* 1. (3 points) What is the time complexity of Sort-Array?

The time complexity of the Sort-Array algorithm can be deduced from the total number of comparisons (n)=n(n+1). This expression simplifies in terms of Big O notation, which represents the algorithm's time complexity. This means the Sort-Array algorithm has a quadratic time complexity as n increases.

* **Appendix**: Grading: What is an OBVIOUS and CLEAR LINK?
* Here is an example to explain what an **obvious and clear link** is and how we grade your work.
* Consider the following problem:
* "(100 points) John travels from Auburn to Atlanta in his car at a speed of 60 mph. Leaving at 8am, at what time will John reach Atlanta".
* Here are the answers of three students and their scores:
* **Student 1** answers: "9:48am". Student 1 will get 25 points.
* **Student 2**answers : "John will reach Atlanta at 9:48am". Student 2 will get 25+15 = 40 points
* **Student 3** answers: "The time t to travel a distance d at speed v is equal to d/v = d/60mph. The problem does not provide the distance d from Auburn to Atlanta. Based on GoogleMaps, the distance from Auburn to Atlanta is approximately 108 miles (**document is attached**).
* 
* Therefore, the time t = 108 miles/60mph \* 60 minutes/hour= 108 minutes. Since John left at 8am, he will then reach Atlanta at 8am + 108 minutes = 8 am + 60 minutes + 48 minutes = 9:48".
* **Student 3** will get 25 + 15 + 60 = 100 points
* Do you see the **direct** **link** going from the data provided in the question to the final answer, using general knowledge/formula and documents?.... Can you now solve the following problem and get 100 points?
* "(100 points) Alice travels from Auburn to Atlanta in her car at a speed of 60 mph. Leaving at 8am, at what time will Alice reach Atlanta assuming that she had a flat tire that delayed her 30 minutes".

1. See **Appendix** about what an obvious and clear link is. [↑](#footnote-ref-1)