CSE 551 - Foundations of Algorithms

Assignment 1

Q1. Use the definition of **Θ**-notation to prove that $n^2/8 - 10n - 4 = \Theta(n^2)$.

Solution:

According to the definition of **Θ**-notation we can assume that $c_1 n^2 <= n^2/8 - 10n - 4 <= c_2 n^2$

Let's Simplify it by dividing both sides by n^2

=> $c_1 <= 1/8 - 10/n - 4/n^2 <= c_2$

For the right-Hand Side (RHS) to be true:

n is positive, it cannot be zero, so n>=1, and c_2 will get closer to 1/8 with the increasing value of n. So, => $c_2 >= 1/8$.

For the left-Hand Side (LHS) to be true:

$c_1 > 0$, So => $1/8 - 10/n - 4/n^2 > 0$

For the above inequality to satisfy, n > 80 => n>= 81,

So, $n_0 = 81$

Putting n_0 in the equation gives => $1/8 - 10/81 - 4/81^2 = 1/8 - 10/81 - 4/6561 = 49/52488$

=> $c_1 <= 49/52488$

Thus, the Parameters are $c_1 <= 49/52488$, $c_2 >= 1/8$ and $n_0 = 81$

That is, $(49/52488)\, n^2 <= n^2/8 - 10n - 4 <= (1/8)\, n^2$

Hence proved.

Q2. Given the recurrence

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 3T(n/3) + n & otherwise \end{cases}$$

(1) Use the iteration method to devise the solution to the recurrence.

(2) Use the recursion-tree method to devise the solution to the recurrence.

Solution:

Part 1: Iteration method

To simplify the derivation let's assume $n = 3^k$, then

$T(n) = 3T(n/3) + n$ => $T(3^k) = 3T(3^k/3) + 3^k$

$T(3^k)/3^k = 3T(3^{k-1})/3^k + 1 = T(3^{k-1})/3^{k-1} + 1$

$T(3^k)/3^k = T(3^{k-1})/3^{k-1} + 1$

So, now I can see $T(3^{k-1})/3^{k-1} = T(3^{k-2})/3^{k-2} + 1$

$T(3^k)/3^k = (T(3^{k-2})/3^{k-2} + 1) + 1 \xrightarrow{repeat} = T(3^0)/3^0 + k$

$T(3^k)/3^k = T(1) + k$

As given for n = 1, T(1) = 1

So, $T(3^k)/3^k = 1 + k$

$T(3^k) = (k+1).3^k$ ------- (1)

Now, $n = 3^k$

Taking $\log_3$ on both sides,

$\log_3 n = \log_3 (3^k) \Rightarrow \log_3 n = k$

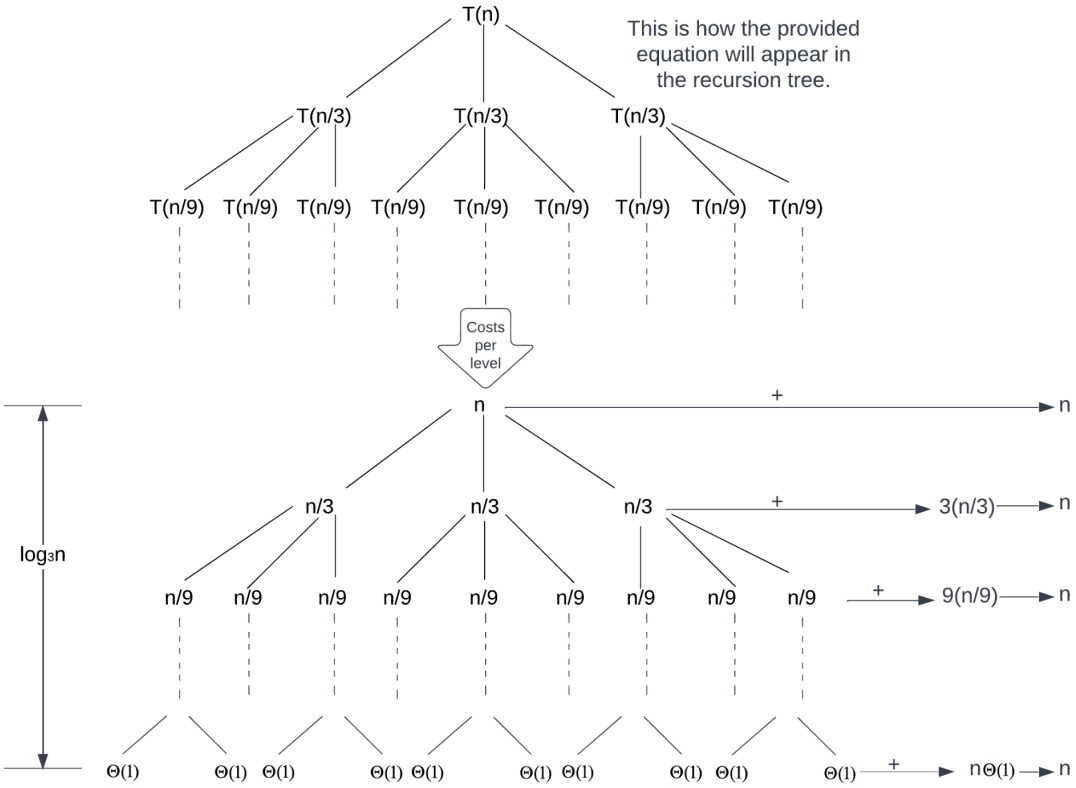
Substituting k in equation (1)

$T(n) = (\log_3 n + 1).n$

$T(n) = n \log_3 n + n \Rightarrow T(n) = \mathbf{\Theta}(n \log_3 n) = \mathbf{\Theta}(n \lg n)$

Hence Proved.

Part 2: Recursion Tree



Height of the tree (h):

We know $3^0 \rightarrow 3^1 \rightarrow 3^2$ - - - $\rightarrow 3^h$ and we reached 3^h in the end

So, In the last call, we can see $\Theta(1) = T(1) = T(n/3^h)$ i.e $(n/3^h)=1$ => So, $3^h = n$

Taking $\log_3$ both sides => $h = \log_3 n$

Total Cost = cost at each level * height of the tree

$= n * \log_3 n = \Theta(n \lg n)$

Hence Proved.

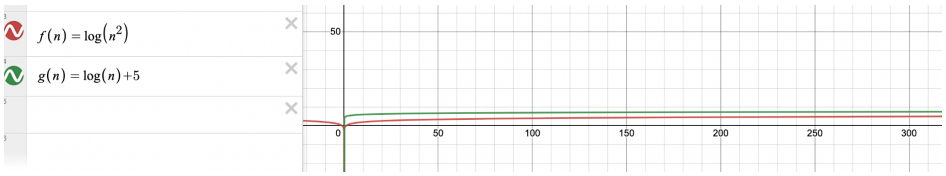
Q3. For each of the following pairs of functions, one of the following relationships holds: f(n) = O(g(n)), f(n) = **Ω**(g(n)), or f(n) =**Θ**(g(n)). Determine which relationship holds and briefly explain your answer.

Solution:

a. f(n) = log n^2; g(n) = log n + 5 => **f(n) = Θ(g(n))**

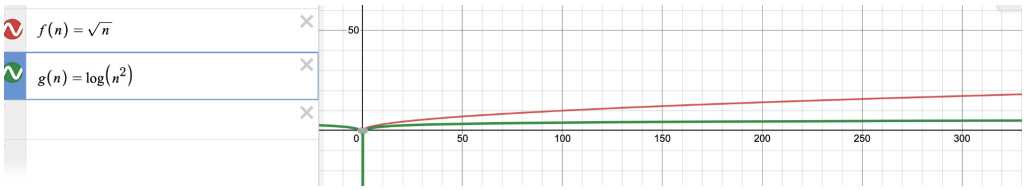
f(n) can be written as (2 log(n)) and g(n) = log(n) + 5. This means their growth is almost the same if we ignore the constants. So for some values of the constant c, (c g(n)) will be the upper bound, and for others, it will be the lower bound of f(n).

(Using Theorem For any 2 functions f(n) and g(n), f(n) = **Θ**(g(n)), iff f(n) = O(g(n)) and f(n) = **Ω**(g(n)))



b. f(n) = √n; g(n) = log n^2 => **f(n) = Ω(g(n))**

g(n) can be written as (2 log(n)). The growth of g(n) is not as fast as f(n). The √n functions grow asymptotically faster than log(n).

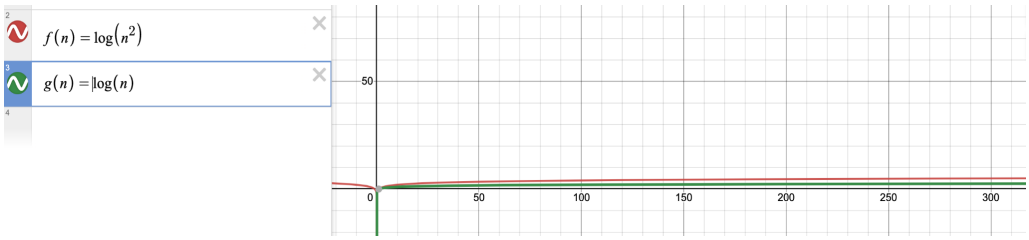


c. f(n) = log n^2 ; g(n) = log n => **f(n) = Θ(g(n))**

f(n) can be written as (2 log(n)) and g(n) = log(n). Their growth is almost similar.

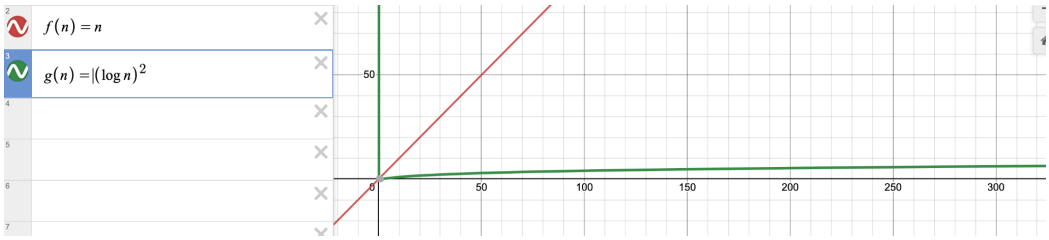
If we take c1.g(n), then for some values of c1, g(n) can become the upper bound of f(n) and for some values (c_2), it can also become the lower bound of f(n), So we can enclose f(n) using some values of c1 and c2 => c1.g(n) <= f(n) <= c2.g(n).

(Using Theorem For any 2 functions f(n) and g(n), f(n) = **Θ**(g(n)), iff f(n) = O(g(n)) and f(n) = **Ω**(g(n)))



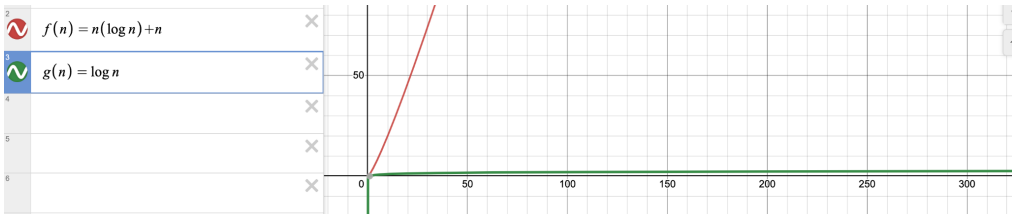
d. f(n) = n; g(n) = $\log^2$ n => **f(n) = Ω(g(n))**

g(n) although a square of log(n) will grow slower than n. So, g(n) is the lower bound of f(n). It is only at the start that g(n) is greater than f(n) where n = 0.



e. f(n) = n log n + n; g(n) = log n => **f(n) = Ω(g(n))**

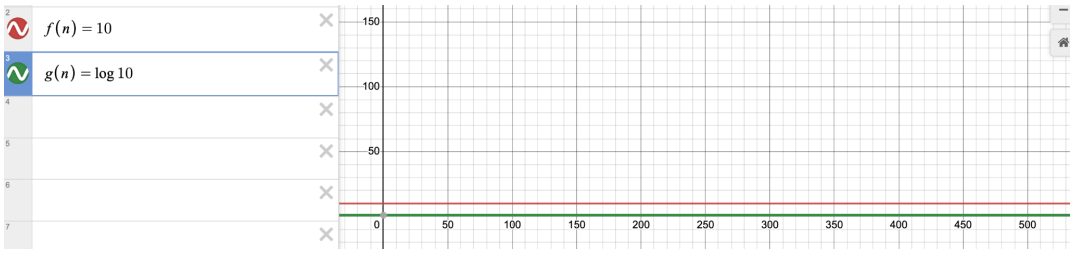
f(n) grows faster than g(n) because it has a log(n) multiplied by n along with another function which is being added to f(n). so , asymptotically, growth of f(n) is faster than g(n). Thus, g(n) represents the lower bound of f(n).



f. f(n) = 10; g(n) = log 10 => **f(n) = Θ(g(n))**

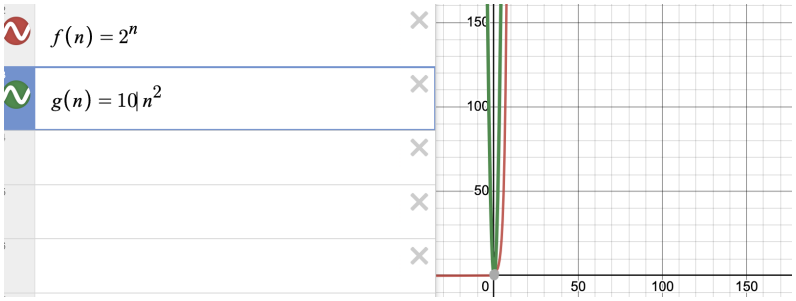
Both the functions are some constant lines to the x-axis. So, for some values of the constant c, (c g(n)) will be the upper bound, and for others, it will be the lower bound of f(n).

(Using Theorem For any 2 functions f(n) and g(n), f(n) = **Θ**(g(n)), iff f(n) = O(g(n)) and f(n) = **Ω**(g(n)))



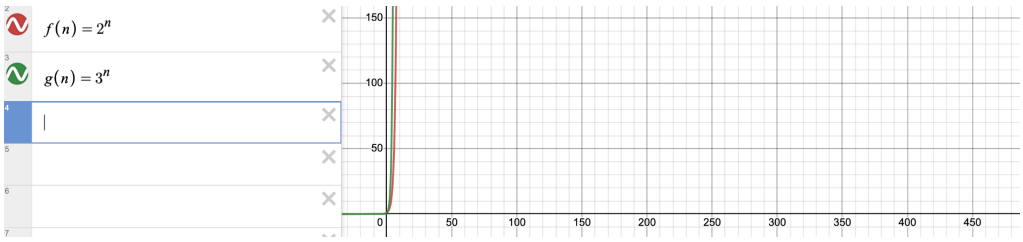
g. f(n) = 2^n; g(n) = $10n^2$ => **f(n) = Ω(g(n))**

f(n) is growing exponentially. And g(n) growth is slower than f(n).



h. f(n) = 2^n; g(n) = 3^n => **f(n) = O(g(n))**

The growth of g(n) is faster than f(n) because g(n) has 3 as its base, which is larger than 2. so, for the same value of n, g(n) will be greater than f(n).



Q4. Consider sorting n numbers stored in array A. You want to sort the numbers by first finding the largest element of A and putting it in the last entry of another array B. Then find the second largest element of A and put it in the second to last entry of B. Continue this manner for n elements of A.

4.1 Write pseudocode for this algorithm and give the number of times that each instruction will be executed.

4.2 Give the best-case running time of the algorithm.

4.3 Give the worst-case running time of the algorithm.

Solution:

4.1 pseudocode

```
for i := n-1 to 0 do                    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n
        //Assume 0th index is largest element
        maxIdx := 0                     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n
        //find index of largest element
        // this inner loop will run (n-1) first then (n-2), then (n-3) and so on till 1.
        // So, the execution time for inner loop is 1+2+---------+ n-2 + n-1
        // which is equal to (n-1) * (n-1+1)/2 = n*(n-1)/2.
        for j := 1 to i do              - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n*(n-1)/2
                if A[j] > A[maxIdx] do  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n*(n-1)/2
                        maxIdx := j
        //swap for A
        temp := A[maxIdx]               - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n
        A[maxIdx] := A[i]               - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n
        A[i] := temp                    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n
        // Insert the chosen element to Array B
        B[i] := A[i]                    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - n
```

4.2 Best-case running time of the algorithm

Even if the array is sorted/unsorted the inner loop will run from 1 to i to find the largest number from the list because the sorting is done from the back and comparison is done from the front and so Algo has to compare all the elements from 1 to i to find the largest.

So, Best case running time algorithm

$T(n) = c_1\, n + c_2\, n + c_3\, (n*(n-1)/2) + c_4\, (n*(n-1)/2)$

$= c_{12}\, (n) + c_{34}\, (n*(n-1)/2)$

$= c_{12}\, (n) + c_{34}\, (n*(n-1)/2)$

$= c_{12}\, (n) + c_{34}/2\, (n^2-n)$

$= c_{12}\, (n) + c_{340}\, (n^2-n)$

$= c_{340}\, (n^2) + n\, (c_{12} - c_{340})$

$\Rightarrow \mathbf{\Theta}(n^2)$

4.3 Worst-case running time of the algorithm

Even if the array is sorted/unsorted the inner loop will run from 1 to i.

So, Worst case running time algorithm

$T(n) = c_1\, n + c_2\, n + c_3\, (n*(n-1)/2) + c_4\, (n*(n-1)/2)$

$= c_{12}\, (n) + c_{34}\, (n*(n-1)/2)$

$= c_{12}\, (n) + c_{34}\, (n*(n-1)/2)$

$= c_{12}\, (n) + c_{34}/2\, (n^2-n)$

$= c_{12}\, (n) + c_{340}\, (n^2-n)$

$= c_{340}\, (n^2) + n\, (c_{12} - c_{340})$

$\Rightarrow \mathbf{\Theta}(n^2)$

Q5. Use a high-level programming language (Java, C++, or Python) to write a program to. Implement the algorithm that you developed in the previous question. In this question, you will complete the following sub-questions.

5.1 For a given input value n, generate n random numbers between 0 and 999 and store these numbers in array A.

5.2 Implement the sorting algorithm and use the array generated in question 5.1 as the input.

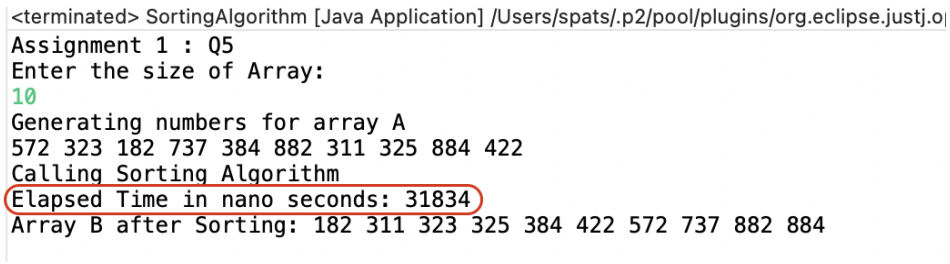
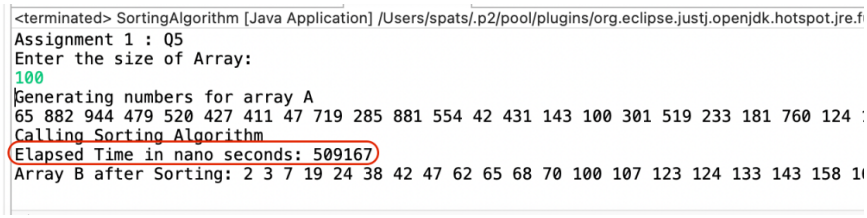
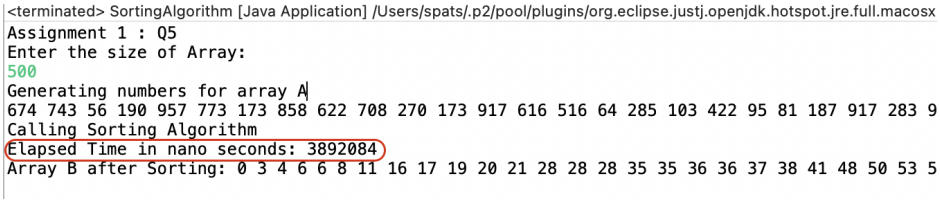
5.3 Print the execution time used for sorting the numbers. Do not include the time for generating the random numbers. Record the following inputs and outputs results in the table.

Solution: ***IDE used = Eclipse***

5.1 – 5.2

```
package assignment1;
import java.util.Random;
import java.util.Scanner;
public class SortingAlgorithm {
	public static void main(String[] args) {
		System.out.println("Assignment 1 : Q5");
		System.out.println("Enter the size of Array: ");
		Scanner sc = new Scanner(System.in);
		int n = sc.nextInt();
		int[] A = new int[n];
		int[] B = new int[n];
		System.out.println("Generating numbers for array A");
		for(int i =0; i<n;i++) {
			Random random = new Random();
			int rnd = (int) (random.nextInt(1000));
			A[i] = rnd;
			System.out.print(A[i]+" ");
		}
		long start = System.nanoTime();
		System.out.println("\nCalling Sorting Algorithm");
		for(int i=n-1; i>=0; i--){
			int maxIdx = 0;
			//find largest element
			for(int j=1; j<=i; j++)
				if(A[j] > A[maxIdx])
					maxIdx = j;
			//swap for A
			int temp = A[maxIdx];
			A[maxIdx] = A[i];
			A[i] = temp;
			// Insert the chosen element to Array B
			B[i] = A[i];
		}
		long end = System.nanoTime();
		System.out.println("Elapsed Time in nano seconds: "+ (end-start));
		System.out.print("Array B after Sorting: ");
		for(int i =0; i<n;i++)
			System.out.print(B[i]+" ");
	}
}
```

5.3 Outputs results in the table

Size	Execution time (ns)
10	31834 
100	509167 
500	3892084 
1000	5577875