Academic Year: <u>2020</u>

Semester 3

Course Code: CS-212L

Course Title: Data structures and Algorithms Lab

CS212L-Data Structure and algorithms

Type of Lab: Open Ended **Weightage:**

CLO 1: CLO's.

State the Rubric	**Cognitive/Understanding**	CLO1	Rubric A

Rubric A: Cognitive Domain

Evaluation Method: GA shall evaluate the students for Question according to following rubrics.

CLO	**0**	**1**	**2**	**3**	**4**
CLO1	Mention Milestones with respect to rubrics				

CS212L-Data Structure and algorithms

Lab 2

Analysis of Algorithms with respect to Time complexity

Objectives: To get familiar with time complexity of Algorithm

Processing steps:

Step 1: what is Time complexity?

Time complexity estimates the time to run an algorithm. It's calculated by counting elementary operations.

Step 2: Why we studied Time complexity?

- We studied time complexity of an algorithm because it help to make algorithm easily to analysis, such that how much it take time to solve the particular problem.
- Time complexity utilize by different data structures to estimate the time
- It is very useful while taking account different data structure to solve the problem while without data structures it takes a lot of time
- Time complexity help to check whether current data structure take how much time to solve the particular problem instead of solving the problem without data structures

Step 3: How time complexity is analysis?

Time complexity is analysis by three operation by

- Big O (O)
- Big omega (Ω)
- Theta (Θ)

Step 4: Analysis of Big O?

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described: ***f(n) = O(g(n)) if there exists a positive integer n_0 and a positive constant c, such that f(n)≤c.g(n) ⍰ n≥n_0 where c is a nonzero constant.***

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Best running time:

The fastest possible running time for any algorithm is O(1), commonly referred to as *Constant Running Time*. In this case, the algorithm always takes the same amount of time to execute, regardless of the input size. This is the ideal runtime for an algorithm, but it's rarely achievable.

But actually it is not the case:

A logarithmic algorithm – O(logn)
Runtime grows logarithmically in proportion to n.

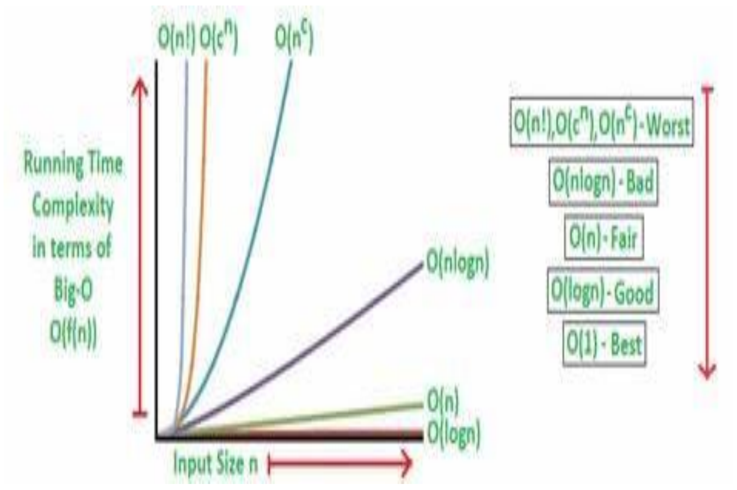
A linear algorithm – O(n)
Runtime grows directly in proportion to n.

A superlinear algorithm – O(nlogn)
Runtime grows in proportion to n.

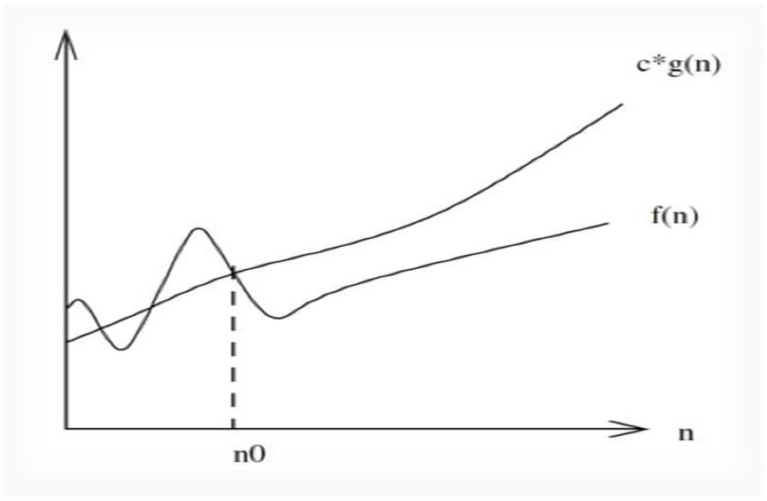
A polynomial algorithm – $O(n^c)$
Runtime grows quicker than previous all based on n.

A exponential algorithm – $O(c^n)$
Runtime grows even faster than polynomial algorithm based on n.

A factorial algorithm – O(n!)
Runtime grows the fastest and becomes quickly unusable for even
small values of n…



Now look on the graph of Big O



constant time complexity:

```
function swap(a, b):
temp = a
a = b
b = temp
```

The time complexity is O(1) as it is independent of input size.

Logarithmic time complexity:

```
def bin_search(data, value):   n = len(data)      #get length of list
  left = 0          #initiate left node
  right = n – 1      #make right node as n-1   while left <= right:      #when left node <= to
right node
    mid=(left + right)//2    #divide in two equal parts
    if value < data[mid]:    #if value less than middle number
      right = mid – 1      #then mid–1value in right node
    elif value > data[mid]:   #if value greater than middle number
      left = mid + 1        #then left node is mid+1
    else:
      return mid          #else return middleif __name__ == '__main__':   data = [10, 20, 30, 40, 50,
60, 70, 80, 90]
   print(bin_search(data, 8))
```

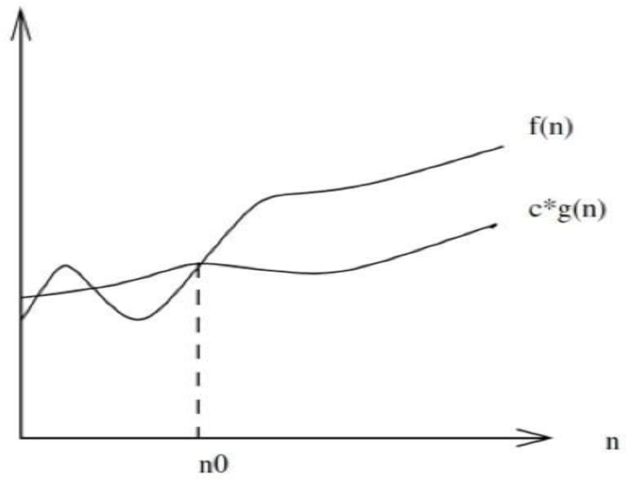
Here you can see in the code that we are dividing the input size at each step in two parts, hence we can conclude that the time complexity here is O(log n).

Step 5: Analysis of big omega:

Omega notation represents the lower bound of the running time of an *algorithm*. Thus, it provides best case complexity of an algorithm.

f(n) = Ω (g(n)) if there exists a positive integer n_0 and a positive constant c, such that $0 \leq c.g(n) \leq f(n)$ ∀ $n \geq n_0$ where c is a nonzero constant.

Graphical view of big omega as follows:



Constant time complexity:

In case of Binary search algorithm we can say that it has its best case as Ω(1), if the number you are finding falls right in the middle.

Step 6: Big-Θ (Big-Theta)

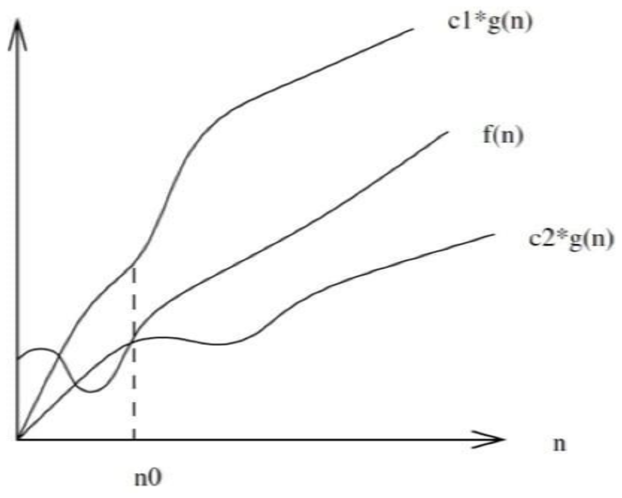
This notation defines a tight bound of an algorithm execution time. So, there is an upper bound and a lower bound and the algorithm execution time would fall within the range.

Mathematical notation:

$f(n) = \theta(g(n))$. Where positive constants n_0, c_1, c_2 are such that at the right of n_0 , the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ i.e.

$$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0}$$

Graphical Representation:



Example: $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Not put this equation in the standard form

$$0 \leq c_1n^2 \leq \frac{1}{2}n^2 \ - 3n \leq \ c_2n^2$$

$$=> \ c_1n^2 \leq \frac{1}{2}n^2 \ - 3n \leq \ c_2n^2$$

Let's do some algebraic operations, we divide the equation with n^2 to make it simpler form

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq \ c_2$$

By doing some practice and putting different values of $c_2 \geq \frac{1}{2}$. by having this value of c_2 our right side of equation is always going to be true when n ≥ 1. Similarly, we can find some value of c_1 and n where left side inequality also holds.

Thus, by having, $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ and n_0 =7, we will come to know that our equation always lies between order of n^2 if we have this property than we ca say that this equation is = $\Theta(n^2)$

In just single line, if we find such value of c_1, c_2 and n0 where the standard equation of Big-Theta satisfied for all values in domain of c_1, c_2 and n_0 then we can say that following function is of order of Θ(g(n)) , here g(n) mean the dominating term of function.

Class activity

- Verify that 2^{n+1} is $O(2^n)$!
- If $f(n) = n^3 + 4n^2$ and $g(n)= n^2$, find the values of c and n such that f(n) is Big-Omega of g(n)
- Prove that $n^2 + 5n + 7 = \Theta(n^2)$

REFRENCES:

Introduction to Algorithms, by CLRS, Third Edition