Data Structures and Algorithm

Lecture

By

Engr Fatima Jaffar

Asymptotic analysis

- Efficiency of an algorithm is measured by:
    1. Time
    2. Space
- Ideal algorithm : accupies least possible time and space
- We will find time complexity

On what basis should we compare the time complexity of data structures

- The time complexity can be compared based on operations performed on them.
- For example:

  Adding a new element in an array vs adding in linked list

How to find the Time Complexity or running time of algorithm?

- The measuring of the actual running time is not practical at all.
- The running time to perform any operation depends on the size of the input
- For example:
- We need to add an element into any array of 100 elements
- vs
- We need to add an element into array of 1000 elements

- Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

How to calculate f(n)

- We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes.
- Now, how to find f(n).

Let's look at a simple example.
$f(n) = 5n^2 + 6n + 12$
where n is the number of instructions executed, and it depends on the size of the input.
When n=1

$$\% \text{ of running time due to } 5n^2 = \frac{5}{5+6+12} * 100 = 21.74\%$$

$$\% \text{ of running time due to } 6n = \frac{6}{5+6+12} * 100 = 26.09\%$$

$$\% \text{ of running time due to } 12 = \frac{12}{5+6+12} * 100 = 52.17\%$$

n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

it is observed that for larger values of n, the squared term consumes almost 99% of the time. As the n^2 term is contributing most of the time, so we can eliminate the rest two terms.

Therefore:

$$f(n) = 5n^2$$

- Here, we are getting the approximate time complexity whose result is very close to the actual result
- this approximate measure of time complexity is known as an Asymptotic complexity.

- asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance

Usually, the time required by an algorithm comes under three types:

Worst case: It defines the input for which the algorithm takes a huge time.
Average case: It takes average time for the program execution.

Best case: It defines the input for which the algorithm takes the lowest time

Asymptotic notation

- The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (O)
- Omega Notation (Ω)
- Theta Notation (θ)

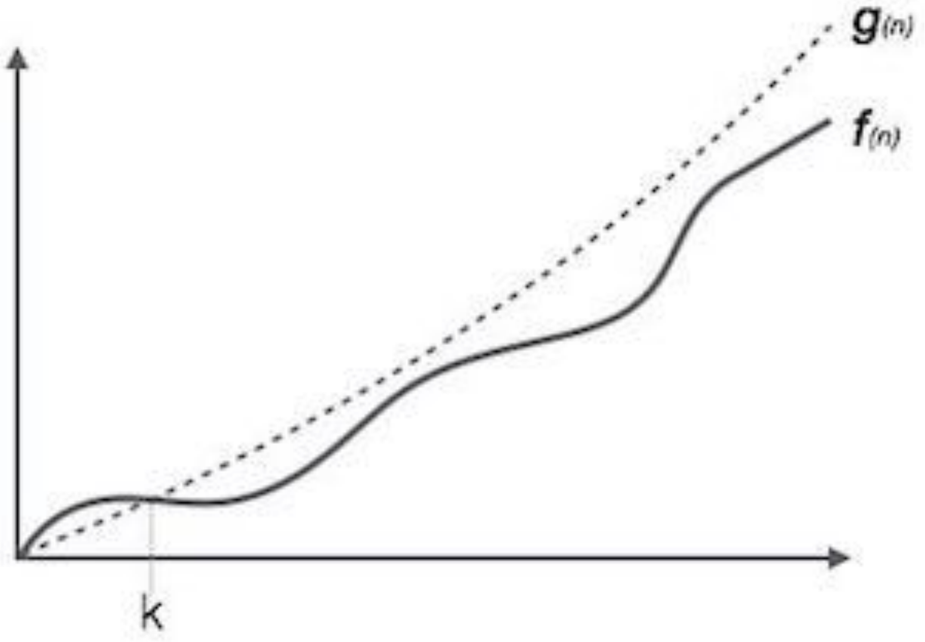
Big O notation (O)

- Measures the performance of an algorithm by providing the order of growth of the function.

- Provides an upper bound on a function which ensures that the function never grows faster than the upper bound.

- It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation.

- It is represented as:



- **For example:**

- If **f(n)** and **g(n)** are the two functions defined for positive integers, then **f(n) = O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

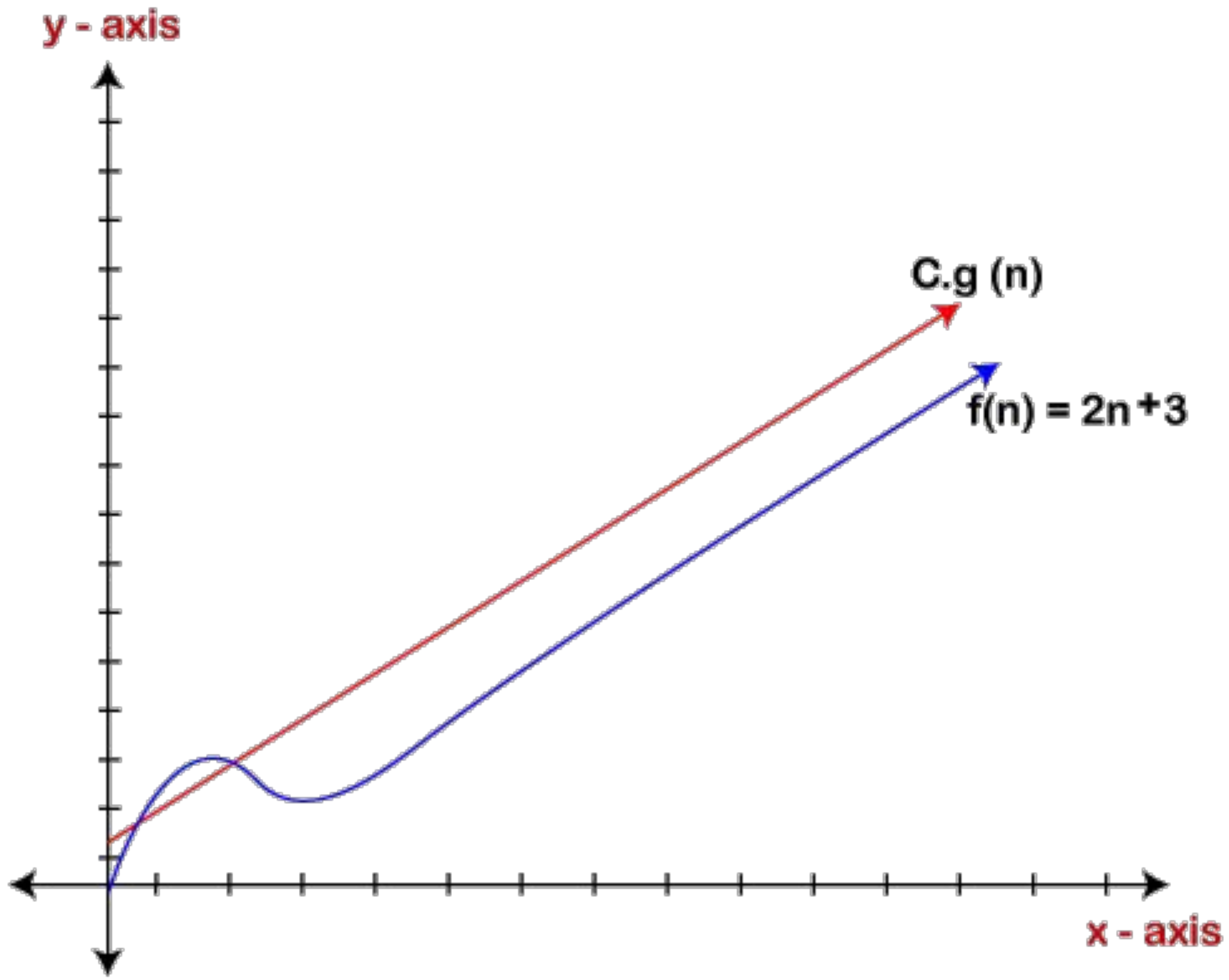
  f(n)≤c.g(n) for all n≥no

- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n)

Example of Big O

: f(n)=2n+3 , g(n)=n

- Now, we have to find **Is f(n)=O(g(n))?**
- To check f(n)=O(g(n)), it must satisfy the given condition:
- **f(n)<=c.g(n)**
- First, we will replace f(n) by 2n+3 and g(n) by n.
- 2n+3 <= c.n
- Let's assume c=5, n=1 then
- 2∗1+3<=5∗1
- 5<=5
- For n=1, the above condition is true.
- If n=2
- 2∗2+3<=5∗2
- 7<=10
- For n=2, the above condition is true.


y - axis
C.g (n)
f(n) = 2n+3
x - axis

Big Omega Notation (Ω)

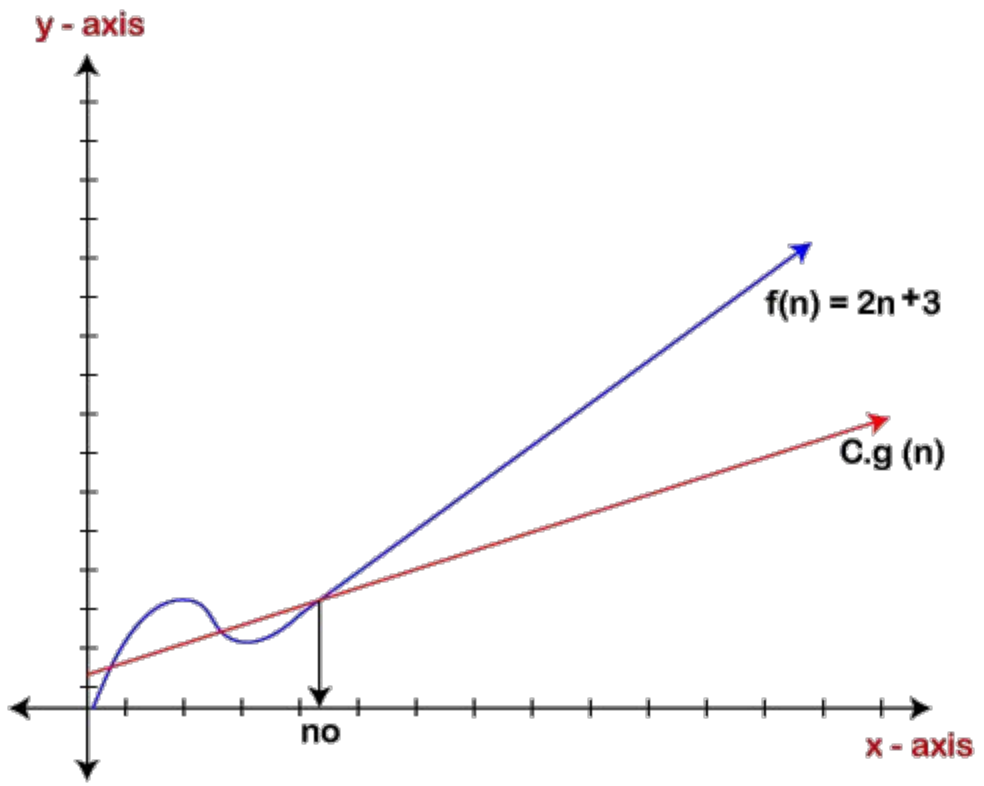
- It basically describes the best-case scenario
- It is the formal way to represent the lower bound of an algorithm's running time.
- It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

- If **f(n)** and **g(n)** are the two functions defined for positive integers, then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)>=c.g(n) for all n≥no and c>0

Example

- If f(n) = 2n+3, g(n) = n,
- Is f(n)= **Ω** (g(n))?
- It must satisfy the condition:
- **f(n)>=c.g(n)**
- To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.
- **2n+3>=c*n**
- Suppose c=1
- **2n+3>=n** (This equation will be true for any value of n starting from 1).
- Therefore, it is proved that g(n) is big omega of 2n+3 function.



y - axis
f(n) = 2n+3
C.g (n)
no
x - axis

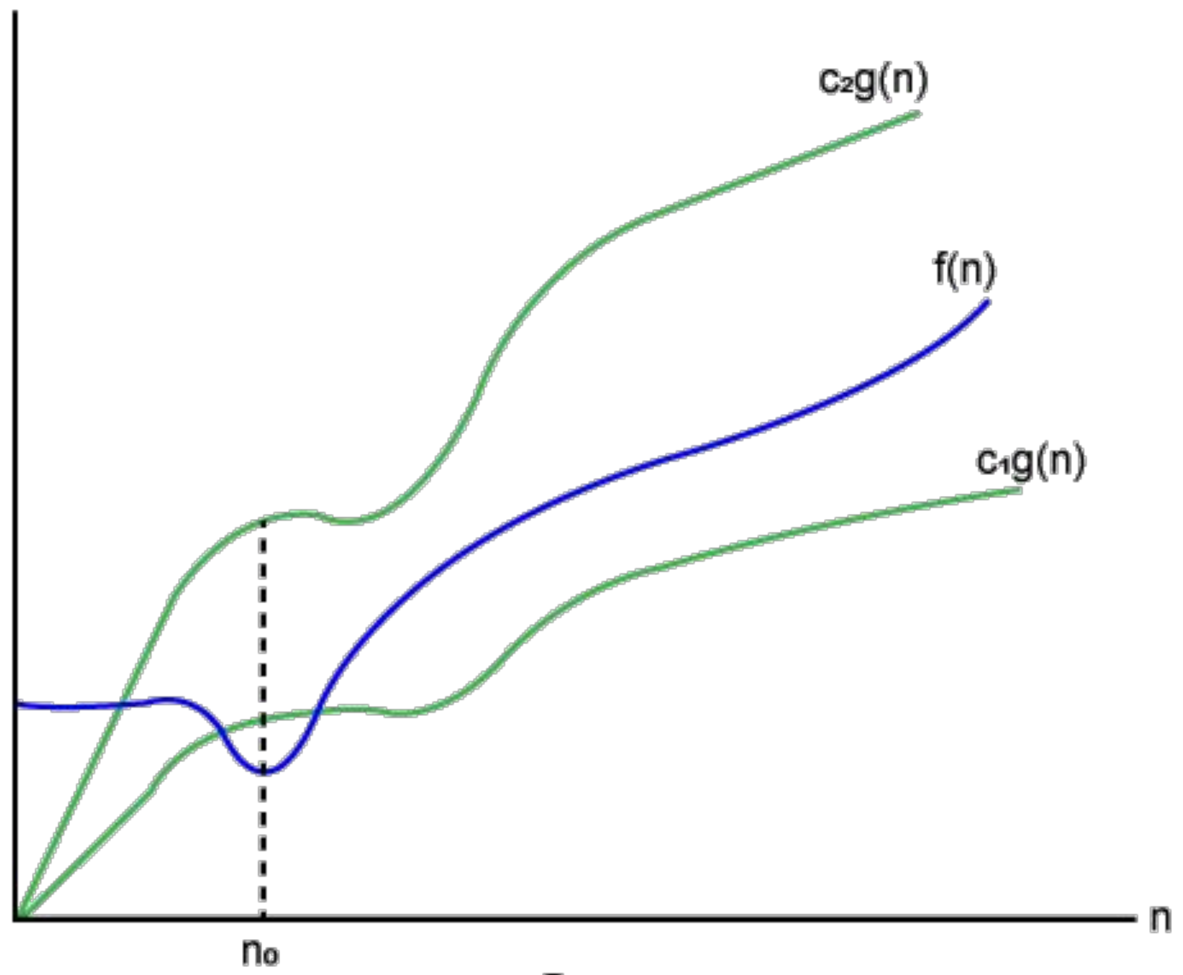
Big theta notation-θ

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.

Big theta notation-θ

- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

- Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:
- **f(n)= θg(n)**
- The above condition is satisfied only if when
- **c1.g(n)<=f(n)<=c2.g(n)**
- where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n).



$f(n) = \Theta(g(n))$

Example

- f(n)=2n+3
  g(n)=n

- As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).
- c1.g(n)<=f(n)<=c2.g(n)
- Replace g(n) by n and f(n) by 2n+3
- c1.n <=2n+3<=c2.n
- if c1=1, c2=2, n=1
- 1*1 <=2*1+3 <=2*1
- **1** <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

- **If n=2**
- 1*2<=2*2+3<=2*2
- 2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)
- Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

Why we have three different asymptotic analysis?

- As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm.

- Suppose we have an array of n numbers, and we want to find the particular element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose, if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst case would be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.