gradeup

Performance Analysis of Algorithms and Recurrences

Performance Analysis of Algorithm and Recurrence
Content:
1. Complexity of algorithm
2. Growth Rate Function
3. Asymptotic Notation
a. Big-OH Notation
b. Big OMEGA Notation
c. Theta Notation
d. Question on Notation
4. Recurrence Relation

COMPLEXITY OF ALGORITHMS
It is suitable to organize algorithms based on the relative amount of time or relative amount of space they require for specify the growth of time/space requirement as a function of the input size.
1. **Time Complexity.** Duration of the program acts as a function of the size of input.
2. **Space Complexity.** Many of what we will be discussing is going to be how structured many algorithms are in terms of time, but some forms of analysis could be done based on how much space an algorithm required to finish its task. This space complexity analysis was analytic in the previous days of calculating when storage space on a computer (both internal and external) was limited. When considering space complexity, algorithms are separated into those that require extra space to do their work and those that work in place. It was not uncommon for programmers to decide an algorithm that was slower just because it worked in place, because there was not enough extra memory for a faster algorithm.
BEST, WORST AND AVERAGE – CASE COMPLEXITY
Using the random – access machine (RAM) model of computation, we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. In RAM model, instructions are achieved one after another, with no concurrent operations. However, to understand how good or bad an algorithm is, we must know how it work s over all instances.
To understand the notations of the best, thrash and average – case complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. These cases of a given algorithm showed what the resource usage is at least, at most and on average, respectively. Generally resource being examine during running time, but it could also be memory or other resource. It is defined by the maximum number of steps taken on any instance of size n. the best – case complexity function defined by the minimum number of steps taken on any instance of size n. finally, the average – case complexity function defined by the average number of steps taken on any instance of size n.
WORST – CASE RUNNING TIME. The conduct of the algorithm with respect to the worst possible case of the input instance. It is an algorithm is an upper bound on the running time for any input. Knowing It gives us a guarantee that the algorithm will never take any longer time.
AVERAGE CASE RUNNING TIME. The expected conduct of the input is randomly drawn from a given distribution. The average – case running time of an algorithm is an estimate of the running time for an "average" input. Computation of a average – case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequence, and the running times for the individual sequences.
It is supposed that all inputs of a given size are equally.

AMORTIZED RUNNING TIME. Time required to execute a sequence of (related) operations is averaged over all the operations performed. It can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. It warranty the mean performance of every operation in the worst case.

WORST – CASE ANALYSIS AND AVERAGE – CASE ANALYSIS

Algorithm often trade efficiency of the average case versus efficiency of the worst case i.e. the well organized program on average might have a particularly bad worst case. We have seen real examples for these when we considered algorithms for searching.

For example:-

- Examine the problem of finding the minimum in a list of elements.

Worst case = O(n) ; Average case = O(n)

- Quick sort

Worst case = $O(n^2)$; Average case = O(n log n)

- Merge sort, Heap sort

Worst case = $O(n^2)$; Average case = $O(n^2)$

- Binary search tree : Search for an element

Worst case = O(n); Average case = O(log n)

GROWTH RATE OF FUNCTIONS

Resources for an algorithm are usually expressed as a function of input. Often this function is messy and difficult to work. To study function growth easily, we reduce the function down to the important part.

Let $f(n) = an^2 + bn + c$

In this function, the n^2 term dominate the function, that is when n gets sufficiently large, the other terms bare factor into the result.

Influence terms are what we are interested in to decrease a function, in this we ignore all constants and coefficients and look at the highest order term in relation to n.

ASYMPTOTIC ANALYSIS

It means a line that likely to converge to a curve, which may or may not finally touch the curve. It is a line that stays within bounds.

It is a shorthand way to write down and talk about 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed. These are known to as 'best case' and 'worst case' scenarios respectively.

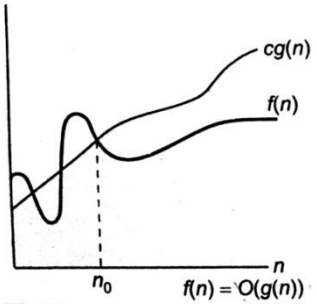
WHY ARE ASYMPTOTIC NOTATIONS IMPORTANT

1. They give a simple assuming of an algorithm's productivity.
2. They allow the comparison of the performance of various algorithms.

ASYMPTOTIC NOTATION

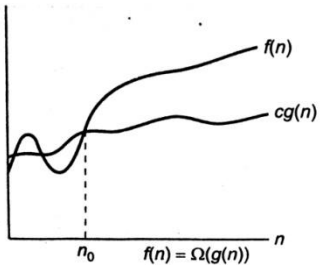
1. BIG-OH NOTATION. Big-oh is the formal method of expression the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possible for take for the algorithm to complete. Legally, for non-negative functions, f(n) and g(n), if there exists an integer n_0 and a constant c > 0 such that for all integers $n > n_0$.

$f(n) \leq c\, g(n)$

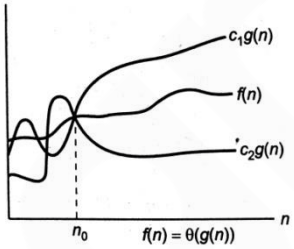


Then f(n) is Big-oh of g(n). This is denoted as
"f(n) ∈ O(g (n))"
i.e., the set of functions which, as n gets large, grow no faster than a constant time f(n).

2. BIG-OMEGA NOTATION. For non-negative function, f(n) and g(n), if there exists an integer n_0 and a constant c > 0 such that for all integers n > n_0, f(n) ≥ c g (n) then f(n) is big omega of g(n). This is denoted as
"f(n) ∈ Ω (g(n))"
This is almost the same definition as Big-on, except that "f(n) ≥ (g(n))", this makes g(n) a lower bound function instead of an upper bound function. It describes the best that can happen for a given data size.



3. THETA NOTATION. The lower and upper bound for the function 'f' is provided by the theta notation (θ). For non-negative functions f(n) and g(n), if there exists an integer n_0, and positive constants c_1 and c_2 i.e., c_1 > 0. And c_2 > 0 such that for all integers n > n_0. c_1g(n) ≤ c_2g(n) then f(n) is theta of g(n). This is denoted as "f(n) ∈ θ (g)" we mean "f is order g".



Example: find the O – notation for the following functions:
a. $f(n) = 5n^3 + n^2 + 6n + 2$ b. $f(n) = 4n^3 + 2n + 3$
c. $f(n) = 10n^2 + 7$ d. $f(n) = 6n^2 + 3n + 2^n$

Solution: Given that :

a. $F(n) = 5n^3 + n^2 + 6n + 2$

For n ≥ 2
$5n^3 + n^2 + 6n + 2 \le 5n^3 + n^2 + 6n + n$
$\le 5n^3 + n^2 + 7n$
For $n^2 \ge 7n$
$5n^3 + n^2 + 7n \le 5n^3 + n^2 + n^2$

$\leq 5n^3 + 2n^2$
For $n^3 \geq 2n^2$
$5n^3 + 2n^2 \leq 5n^3 + n^3 \leq 6n^3$
Thus c = 6, $n_0 = 2$
So, $f(n) = O(n^3)$
b. $f(n) = 4n^3 + 2n + 3$
For $n \geq 3$
$4n^3 + 2n + 3 \leq 4n^3 + 2n + n$
$\leq 4n^3 + 3n$
For $n^3 \leq 3n$
$4n^3 + 3n \leq 4n^3 + n^3$
$\leq 5n^3$
Thus c = 5, $n_0 = 3$
So, $f(n) = O(n^3)$
c. $f(n) = 10n^2 + 7$
For $n \geq 7$
$10n^2 + 7 \leq 10n^2 + n$
For $n \leq n^2$
$10n^2 + n \leq 10n^2 + n^2$
$\leq 11n^2$
Thus c = 11 and $n_0 = 7$
So, $f(n) = O(n^2)$
d. $f(n) = 6n^2 + 3n + 2^n$
For $n^2 \geq 3n$
$2^n + 6n^2 + 3n \leq 2^n + 6n^2 + n^2$
$\leq 2^n + 7n^2$
For $2^n \geq n^2$
$2^n + 7n^2 \leq 2^n + 6 * 2^n$
≤ 8.2
So, $n_0 = 4$, c = 8
Thus, $f(n) = O(2^n)$

Recurrence Relation :
Master Theorem: If recurrence relation is of type
T(n) = aT (n/b) + F(n)
Where
F(n) > 0 ;
a≥ 1;
b>1
then using these 3 rules we can find complexity of the relation & directly write them in the form of 0, Ω , θ.
Case 1 : If $F(n) = 0 (n^{\log_b a} - €)$
Where € > 0 then
$T(n) = \theta (n^{\log_b a})$
Example :
T(n) = 9 T(n/3) + ∞
a = 9 b = 3
F(n) = ∞
Put in case 1

F(n) = $O(n^{\log_b a - €})$
$0(n^{\log_3 9 - €})$
= $n^{\log_3 9}$
= $n^{\log_3 3^2}$
= $n^{2\log_3 3}$
= n^2
So F(n) = $O(n^{2-€})$
Let € = 1 (€>0)
F(n) = O (n^{2-1})
= $O(n^1)$
Case 1 satisfied therefore the complexity of the function is
T(n) = $\theta(n^{\log_b a})$
= θ $(n^{\log_3 9})$
= θ (n^2)

Case 2 :
If F(n) = $\theta(n^{\log_b a})$
T(n) = $(n^{\log_b a}$ X log n)
T(n) = T(2n/3) +1
a = 1; b = 3/2 ; F(n)= 1
First, we check for case 1
F(n) = $O(n^{\log_b a - €})$
1 = $n^{\log_{3/2} 1 - €}$
= $n^{\log_{3/2} 1}$

where power is 1 hen it is 0
so 1 = $n^{0 - €}$
Here we cannot choose any € > 0 which can satisfy the case therefore case 1 is not satisfied
Second: Because case 1 is not satisfied therefore we check case 2.
F(n) = $\theta(n^{\log_b a})$
1 = $\theta(n^{\log_{3/2} 1})$
=$n^{\log_{3/2} 1 \to 0}$
=n^{0}
1 = θ (n^0)
$(\text{anything})^0 = 1$
So 1 = θ(1)
Therefore complexity is
T(n) = $\theta(n^0$ X log n)
= θ(1 X log n)
= θ(log n)

Case 3 : if F(n) = Ω $(n^{\log_b a + €})$
For € > 0
T(n) = θ(F(n))

Example :
T(n) = 3 T(n/4) + nlogn
a = 3
b= 4

F(n) = nlogn
First we check for case 1 & case 2 and if they are not satisfied then no need to check for case 3 directly write answer according to case 3
So , T(n) = θ(nlogn)
Because case 1 & 2 are not satisfied

gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now