Introduction

22 May 2021 16:56

Asymptotic Analysis

- We measure order of growth of time taken by a function/program in terms of input size.
- It doesn't depend upon machine, programming language, test cases, etc.
- It is algorithm specific not test case specific.
- No need to implement, we can analyze algorithms.
- Every athematic or comparison operation is assumed to take a constant time irrespective of input value.

Order of Growth

Order of growth is respective value of n at certain value. Lower the value of n, slower the order of growth.

- A function f(n) is said to be growing faster than g(n) if

  $\lim_{n\to\infty}$ g(n)/f(n) = 0

  OR

  $\lim_{n\to\infty}$ g(n)/f(n) = ∞
- Here g(n) is efficient because its growing slower.

Way to Find Order of Growth

- Ignore lower order terms
- Ignore leading term constant

Example:-

F(n) = $2n^2 + n + 6$

Order of growth: n^2 (Quadratic)

G(n) = $100n + 3$

Order of growth: n (Linear)

Comparison of Order of Growths

$C < \log(\log n) < \log n < n^{1/3} < n^{1/2} < n < n^2 < n^3 < n^4 < 2^n < n^n$

Cases

1. Best Case
- Minimum order of growth.

2. Average Case
- Average order of growth.
- Assumptions are taken.

3. Worst Case
- Maximum order of growth.

Asymptotic Notations

These are the mathematical notation to represent order of growth of any mathematical function.

1. Big O
- Exact or upper bound order of growth
- If O(n) ---> It means linear time or less than linear time.

2. Theta θ
- Exact order of growth

3. Omega Ω
- Exact or lower bound order of growth.
- If Ω(1) ---> It means either constant or any value higher than constant.

Big O Notation

We say f(n) = O(g(n)) iff there exist constants c and n0 such that f(n) <= c g(n) for all n >= n0.

- It is used to represent exact or upper bound order of growth.
- Mostly used for upper bound order of growth.
- Big O notation can be find out by the same way as we find order of growth.

Example:-

$3n + 10\ n\ log\ n + 3$

Big O notation :- O(n log n)

$1000m^2 + 2000mn + 30m + 20n$

Big O notation :- $O(m^2 + mn)$

Omega Notation

f(n) = Ω(g(n)) iff there exist constants c (where c > 0) and n0 (where n0 >= 0) such that cg(n) <= f(n) for all n >= n0.

- It is used to represent exact or lower bound order of growth.
- Mostly used for lower bound order of growth.
- If f(n) = O(g(n))
  Then, g(n) = Ω(f(n))

Theta Notation

f(n) = θ(g(n)) iff there exist constants c1, c2 (where c1 > 0 and c2 > 0) and n0 (where n0>=0) such that c1g(n) <= f(n) <= c2g(n)

- It is used to represent the exact bound order of growth.
- Theta notation is precise.
- If $f(n) = \theta(g(n))$
  Then, f(n) = O(g(n)) and f(n) = Ω(g(n))
  And g(n) = O(f(n)) and g(n) = Ω(f(n))

Example:-

2000n + 2logn

Theta notation :- $\theta(n)$

Complexity Analysis of Loops

1. Common loops

- If incrementation in loop is addition or subtraction of a numerical constant (i.e., i = i + c or i = i - c) then loop run for (n/c) times. So time complexity is θ(n).
- If incrementation of loop is multiplication or division of a numerical constant (i.e., i = i*c or i = i/c) then loop run for ($\log_c n$) times. So time complexity is θ(log n).
- If incrementation of loop is exponential with a numerical constant (i.e., i = pow(i , c)) then loop run for ($\log_c \log_c n$) times. So time complexity is θ(log(log n)).

2. Multiple Loops

- **Subsequent Loops:** If multiple loops in a function are independent (i.e., non-nested loops) then the time complexity of loop is added, but only highest order growth complexity out all loop is consider as overall complexity of said function.

  Example:-

  If time complexity of three independent loops of a function are θ(n), θ(log n) and θ(1) respectively. Then the time complexity of fucntion is θ(n).

- **Nested Loops:** If multiple loops in a function are dependent (i.e., nested loops) then the time complexity of loop is multiplied, and the resulted value is the time complexity of the function.

  Example:-

  If time complexity of three dependent loops of a function are θ(n), θ(log n) and θ(1) respectively. Then the time complexity of fucntion is θ(n * log n * 1) which is θ(n log n).

- **Mixed Loops:** If multiple loops in a function are both dependent and independent (i.e., multiple nested loops are independent to each other) then the time complexity of each dependent loop is multiplied with their respective loop and the resulted complexity is added with independent loops, but only highest order growth complexity out of all resulted loops is considered as overall complexity of said function.

  Example:-

  If time complexity of two dependent loops of a function are θ(n) and θ(log n), and time complexity of two more dependent loops of that function are θ(n) and θ(n). [Both nested loops are independent of each other]. Then the time complexity of first set of nested loop is θ(n log n) and second set is θ(n^2). So the time complexity of the fucntion is θ(n^2).

- **Different Input:** If same situation of mixed loop is present in a function, but both the independent loops have different inputs then the time complexity of each dependent loop is multiplied with their respective loop and the resulted complexity is added, and the resulted value is the time complexity of the function.

  Example:-

  If time complexity of two dependent loops of a function are θ(n) and θ(log n), and time complexity of two more dependent loops of that function are θ(m) and θ(m). [Both nested loops are independent of each other]. Then the time complexity of first set of nested loop is θ(n log n) and second set is θ(m^2). So the time complexity of the fucntion is θ($n \log n + m^2$).

Recurrence Relation

Recurrence relation of a function is the time taken T(n). T(n) is submission of each recursive calls to the function and complexity of other operations in the function (i.e., loops or simple constant time operations).

Example:-

```
void fun(int n)
{
    if (n <= 0)
    {
        return;
    }
    print ("GfG");
    fun(n/2);
    fun(n/2);
}
```

Recurrence Relation:-

For n > 0

$T(n) = T(n/2) + T(n/2) + \theta(1)$

$T(n) = 2\,T(n/2) + \theta(1)$

For n <=0 [Base Case]

$T(n) = \theta(1)$

OR

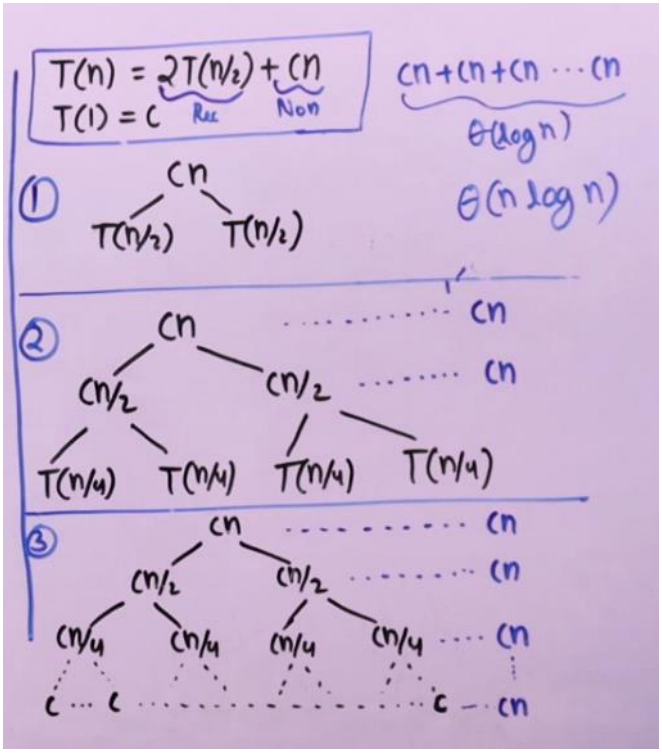
$T(0) = \theta(1)$ [if n is considered +ve]

- In recursion every function calls is stored in the function call stack. And it waits for other function calls to finish so they can get the control back to them and resumes.
  The very first function call sits in the bottom of the stack and the very last at top.

Recursion Tree Method (for solving recurrences)

1. We consider the recursion tree and compute the total work done.
2. We write non-recursive part as root of the tree and write the recursive part as children.
3. We keep expanding until we see a pattern.

Example:-



Complexity of each cn is θ(log n)
So the complexity of all cn is θ(n log n)

Space Complexity

- It is defined as order of growth of memory space (or ram space) in term of input size.
- Space complexity doesn't depend upon the input value as long as it doesn't affect the number of variables (and its not a arrays or variable).
- Space complexity is represented same as asymptotic notations.

Auxiliary Space

- It is the order of growth of extra space or temporary space in terms of input size.
- Auxiliary space is mostly used instead of space complexity, because it gives more information.
- Auxiliary space is the space created in memory for other than input output.
- Example:-

  Bubble Sort didn't require extra memory space for sorting operation. So auxiliary space for bubble sort is $\theta(1)$.

  Merge Sort requires an extra array and hence needs extra memory space for sorting operations. So auxiliary space for bubble sort is $\theta(n)$.
- Auxiliary space for a recursion relation is equivalent to the height of the recurrence tree.