UNIT I ROLE OF ALGORITHMS IN COMPUTING & COMPLEXITY ANALYSIS

Algorithms – Algorithms as a Technology – Time and Space complexity of algorithms – Asymptotic analysis – Average and worst-case analysis – Asymptotic notation – Importance of efficient algorithms – Program performance measurement – Recurrences: The Substitution Method – The Recursion – Tree Method – Data structures and algorithms.

ASYMPTOTIC ANALYSIS – AVERAGE AND WORST-CASE ANALYSIS

Worst Case Analysis (Usually Done)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of linear search would be Θ(n).

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \Theta(n)$$

Best Case Analysis (Bogus)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant(not dependent

on n). So time complexity in the best case would be

(1). Most of the time, we do worst-case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm. The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, Algorithms may take years to run. For some algorithms, all the cases are asymptotically the same, i.e., there are no worst and best cases. For example, Merge Sort.

Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the array into two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

ASYMPTOTIC NOTATION

The main idea of asymptotic analysis is to measure the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

1. Θ notation
2. Big O notation
3. Ω notation

Θ Notation:

The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants.

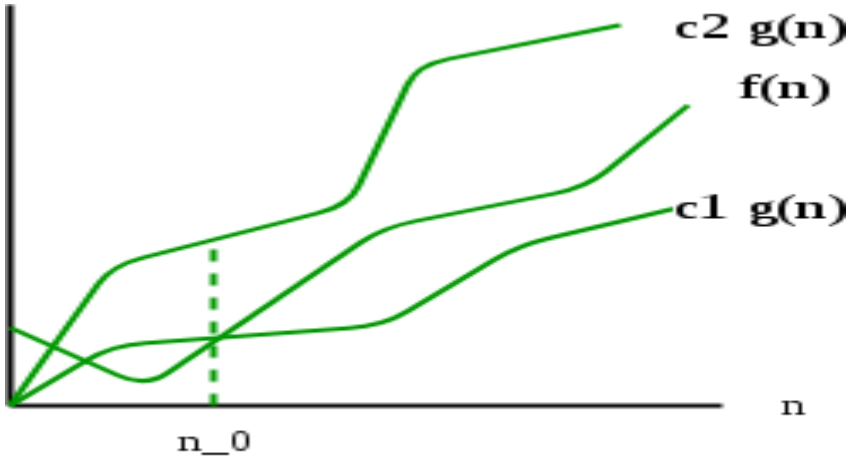
For example, consider the following expression. $\mathbf{3n^3 + 6n^2 + 6000 = \Theta(n^3)}$

Dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <=

c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



f(n) = theta(g(n))

Big O Notation

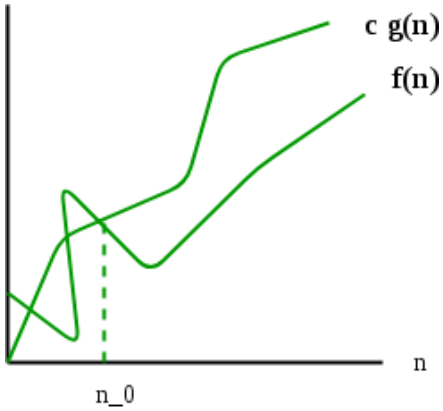
The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst-case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= f(n) <= c*g(n) for all n >= n0}



f(n) = O(g(n))

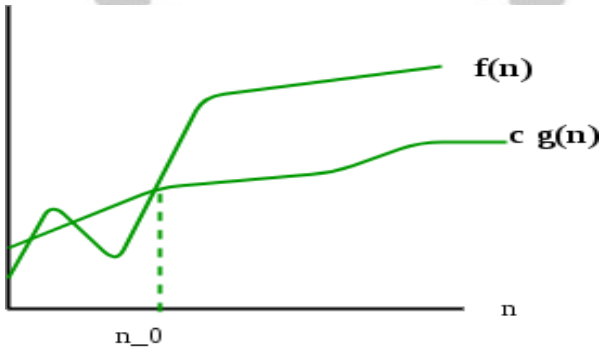
Ω Notation

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation can be useful when we have a lower bound on the time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst- case and sometimes in the average case.



f(n) = Omega(g(n))

Properties of Asymptotic Notations

As we have gone through the definition of these three notations let's now discuss some important properties of those notations.

1. General Properties:

If f(n) is O(g(n)) then a*f(n) is also O(g(n)), where a is a constant.

Example: $f(n) = 2n^2+5$ is $O(n^2)$

then $7*f(n) = 7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

2. Transitive Properties:

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if $f(n) = n$, $g(n) = n^2$ and $h(n)=n^3$

n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .

If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

3. Reflexive Properties:

Reflexive properties are always easy to understand after transitive.

If f(n) is given then f(n) is O(f(n)). Since MAXIMUM VALUE OF f(n) will be f(n) itself! Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.

Example: $f(n) = n^2$; $O(n^2)$ i.e O(f(n))

Similarly, this property satisfies both Θ and Ω notation.

We can say that:

If f(n) is given then f(n) is Θ(f(n)). If f(n) is given then f(n) is Ω (f(n)).

If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)) ; where a is a constant. If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)) ; where a is a constant.

4. Symmetric Properties:

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .

Example: $f(n) = n^2$ and $g(n) = n^2$ then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

5. Transpose Symmetric Properties:

If f(n) is O(g(n)) then g(n) is Ω (f(n)).

Example: $f(n) = n$, $g(n) = n^2$ then n is $O(n^2)$ and n^2 is Ω (n)

This property only satisfies O and Ω notations.