**Analysis of Algorithms**

**Asymptotic Analysis**

***Given two algorithms for a task, how do we find out which one is better?***

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.  
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

[Asymptotic Analysis](http://en.wikipedia.org/wiki/Asymptotic_analysis) is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and the other way is Binary Search (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer **A** and Binary Search on a slow computer **B** and we pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds. Let’s say the constant for **A** is 0.2 and the constant for **B** is 1000 which means that A is 5000 times more powerful than B. For small values of input array size n, the fast computer may take less time. But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after a certain value of input size.

Here are some running times for this example:  
**Linear Search running time in seconds on A**: 0.2 \* n  
**Binary Search running time in seconds on B**: 1000\*log(n)

------------------------------------------------

|n | Running time on A | Running time on B |

-------------------------------------------------

|10 | 2 sec | ~ 1 h |

-------------------------------------------------

|100 | 20 sec | ~ 1.8 h |

-------------------------------------------------

|10^6 | ~ 55.5 h | ~ 5.5 h |

-------------------------------------------------

|10^9 | ~ 6.3 years | ~ 8.3 h |

-------------------------------------------------

**Worst, Average and Best Cases**

We discussed how asymptotic analysis overcomes the problems of naive way of analyzing algorithms. Now we will take an example of Linear Search and analyze it using asymptotic analysis.

We can have three cases to analyze an algorithm:   
1) The Worst Case   
2) Average Case   
3) Best Case

Let us consider the following implementation of Linear Search.

**Worst Case Analysis (Usually Done)**   
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes 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() functions 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 total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are [uniformly distributed](http://en.wikipedia.org/wiki/Uniform_distribution_%28discrete%29) (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity. 

Average Case Time =

analysis1

= analysis2

= Θ(n)

**Best Case Analysis (Bogus)**  
In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes 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 Θ(1)

Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

The average case analysis is not easy to do in most of the 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, an algorithm may take years to run.  
For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example,[Merge Sort](http://en.wikipedia.org/wiki/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 occur when the pivot elements always divide array in 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 Notations**

We have discussed [Asymptotic Analysis](https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/), and[Worst, Average](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/),[and Best Cases of Algorithms](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/). The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don’t depend on machine-specific constants and doesn’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.

thetanotation

**1) Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.   
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.   
3n3 + 6n2 + 6000 = Θ(n3)

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.

BigO

**2) 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 best case and quadratic time in 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 upper bound on 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}

BigOmega

**3) 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 lower bound on time complexity of an algorithm. As discussed in the previous post, the [best case performance of an algorithm is generally not useful](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), 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 a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

**Properties of Asymptotic Notations :**   
As we have gone through the definition of this 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) = 2n²+5 is O(n²)   
     then 7\*f(n) = 7(2n²+5) = 14n²+35 is also O(n²) .

     Similarly this property satisfies for both Θ and Ω notation. 

     We can say   
     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.

**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) = n² and h(n)=n³  
    n is O(n²) and n² is O(n³)  
    then n is O(n³)

   Similarly this property satisfies for 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) = n² ; O(n²) i.e O(f(n))

      Similarly this property satisfies for 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)).

**4. Symmetric Properties :**

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

      Example: f(n) = n² and g(n) = n²   
      then f(n) = Θ(n²) and g(n) = Θ(n²) 

**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) = n²   
      then n is O(n²) and n² is Ω (n)

**This property only satisfies for O and Ω notations**.

**6. Some More Properties :**

     1.) If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))

     2.) If f(n) = O(g(n)) and d(n)=O(e(n))   
          then f(n) + d(n) = O( max( g(n), e(n) ))   
          *Example:* f(n) = n i.e O(n)   
                         d(n) = n² i.e O(n²)   
                         then f(n) + d(n) = n + n² i.e O(n²)

      3.) If f(n)=O(g(n)) and d(n)=O(e(n))   
           then f(n) \* d(n) = O( g(n) \* e(n) )   
           *Example:* f(n) = n i.e O(n)   
           d(n) = n² i.e O(n²)   
                      then f(n) \* d(n) = n \* n² = n³ i.e O(n³)

**Time Complexities of all Sorting Algorithms**

Efficiency of an algorithm depends on two parameters:

1. Time Complexity

2. Space Complexity

**Time Complexity:** Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor’s speed, etc.

**Space Complexity:** Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n).

One important thing here is that in spite of these parameters the efficiency of an algorithm also depends upon the **nature**and **size of**the **input.**

Following is a quick revision sheet that you may refer at last minute

| **Algorithm** | **Time Complexity** | | |  |
| --- | --- | --- | --- | --- |
|  | **Best** | **Average** | **Worst** |  |
| [Selection Sort](http://geeksquiz.com/selection-sort/) | Ω(n^2) | θ(n^2) | O(n^2) |  |
| [Bubble Sort](http://geeksquiz.com/bubble-sort/) | Ω(n) | θ(n^2) | O(n^2) |  |
| [Insertion Sort](http://geeksquiz.com/insertion-sort/) | Ω(n) | θ(n^2) | O(n^2) |  |
| [Heap Sort](http://geeksquiz.com/heap-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |  |
| [Quick Sort](http://geeksquiz.com/quick-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n^2) |  |
| [Merge Sort](http://geeksquiz.com/merge-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |  |