Introduction

▼ Asymptotic Notation

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. O 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.

$$3n^3 + 6n^2 + 6000 = O(n^3)$$

Dropping lower order terms is always fine because there will always be a n0 after which O(n3) has higher values than O(n2) irrespective of the constants involved.

For a given function g(n), we denote O(g(n)) as:

> O(g(n)) = {f(n): 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >=n0}

where c1, c2, and n0 are positive constants.

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.

▼ 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 O 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 $O(n^2)$.
2. The best case time complexity of Insertion Sort is $O(n)$.

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): 0 <= f(n) <= c*g(n) for all n >= n0}

where c and n0 are positive constants

▼ 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. The Omega notation is the least used notation among all three.

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

> $\Omega(g(n))$ = {f(n): 0 <= c*g(n) <= f(n) for all n >= n0}

where c and n0 are positive constants.

▼ Worst, Average and Best Case Time Complexities

It is important to analyze an algorithm after writing it to find it's efficiency in terms of time and space in order to improve it if possible.

When it comes to analyzing algorithms, the asymptotic analysis seems to be the best way possible to do so. This is because asymptotic analysis analyzes algorithms in terms of the input size. It checks how are the time and space growing in terms of the input size.

Let’s take an example of Linear Search and analyze it using Asymptotic analysis. We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Below is the algorithm for performing linear search:

```
// Linearly search x in arr[].
// If x is present then return the index,
// otherwise return -1
search(int arr[], int n, int x)
{
    int i;
    for (i=0;i<n;i++){
      if(arr[i]==x)
    }
    return -1;
 }

 //Driver program to test above functions
 int main(){
  int arr[]={2,8,12,9};
```

```
 int x=12;
 int n=sizeof(arr)/sizeof(arr[0]);
 printf("%d is present in %d index",x,search(arr,n,x));
 getchar();
 return 0;
}
```

Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes the 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 $O(N)$, where N is the number of elements in the array.

Average Case Analysis

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 (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.

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

$$Average\ Case\ Time\ = \frac{\Theta((n+1)(n+2)/2)}{(n+1)}$$

$$Average\ Case\ Time\ = \Theta(n)$$

▼ Best Case Analysis

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 O(1)

▼ Important Points:

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

▼ Analysis of Loops

We have already discussed *Asymptotic Analysis* , *Worst, Average and Best Cases* and *Asymptotic Notations*. In this post, analysis of iterative programs with simple examples is discussed.

▼ 1. $O(1)$:

Time complexity of a function (or set of statements) is considered as $O(1)$ if it doesn't contain loop, recursion and call to any other non-constant time function.

For example swap() function has $O(1)$ time complexity. A loop or recursion that runs a constant number of times is also considered as $O(1)$. For example the

following loop is $O(1)$.

```
// Here c is a constant
   for (int i = 1; i <= c; i++) {
        // some O(1) expressions
   }
```

▼ 2. $O(n)$:

Time Complexity of a loop is considered as $O(n)$ if the loop variables is incremented / decremented by a constant amount. For example following functions have $O(n)$ time complexity.

```
// Here c is a positive integer constant
   for (int i = 1; i <= n; i += c) {
        // some O(1) expressions
   }

   for (int i = n; i > 0; i -= c) {
        // some O(1) expressions
   }
```

▼ 3. $O(n^c)$:

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity.

```
for (int i = 1; i <=n; i += c) {
       for (int j = 1; j <=n; j += c) {
          // some O(1) expressions
       }
   }

   for (int i = n; i > 0; i -= c) {
       for (int j = i+1; j <=n; j += c) {
          // some O(1) expressions
   }
```

▼ 4. $O(logn)$:

Time Complexity of a loop is considered as $O(logn)$ if the loop variables is divided / multiplied by a constant amount.

For example Binary Search(refer iterative implementation) has $O(logn)$ time complexity. Let us see mathematically how it is $O(logn)$. The series that we get in first loop is $1, c, c^2, c^3, ..., c^k$. If we put k equals to $log_c n$, we get $c^{log_c n}$ which is n.

```
 for (int i = 1; i <=n; i *= c) {
       // some O(1) expressions
   }

   for (int i = n; i > 0; i /= c) {
       // some O(1) expressions
   }
```

▼ 5. $O(loglogn)$:

Time Complexity of a loop is considered as $O(loglogn)$ if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
   for (int i = 2; i <=n; i = pow(i, c)) {
       // some O(1) expressions
   }

   // Here fun() is function to find square root
   // or cuberoot or any other constant root
   for (int i = n; i > 1; i = fun(i)) {
       // some O(1) expressions
   }
```

▼ How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
        // some O(1) expressions
   }

   for (int i = 1; i <=n; i += c) {
        // some O(1) expressions
   }
```

Time complexity of above code is $O(m) + O(n)$ which is $O(m + n)$. If $m == n$, the time complexity becomes $O(2n)$ which is $O(n)$.

▼ How to calculate time complexity when there are many if, else statements inside loops?

As discussed earlier, the worst-case time complexity is the most useful among best, average and worst. Therefore we need to consider the worst case. We evaluate the situation when values in if-else conditions cause a maximum number of statements to be executed.

For example, consider the linear search function where we considered the case when an element is present at the end or not present at all.

When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.