**Analysis of Algorithms**

**I. Introduction**

* **What is an algorithm?**

An algorithm is the list of instructions and rules that a computer needs to do to complete a task.

Ex: Find the maximum value in array A

S1: Max ← A[0], i ← 0

S2: i ← i + 1

S3: If i > n, move to step 5

S4: If A[i] > Max, Max ← A[i], go back to step 2

S5: end.

* **Why performance analysis?**
* Algorithm analysis helps determine which algorithms are effective in terms of space and time, …
* When we work with large programs, we will have a good sense of where major slowdowns are likely to cause bottlenecks, and where more attention should be paid to get the largest improvements.

**II. Three cases to analyze an algorithm**

Let us consider the following implementation of Linear Search

****

**1. The Worst Case (O)**

*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 is not present in the array (x in the above code). When x is not present, the **search**() functions compare it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(n).

**2. The Average Case (**Θ)

*In average case analysis, we take all possible inputs and calculate computing time for all the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know distribution of cases*.

For *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).

**3. The Best Case (Ω)**

*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 *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).

Lower bound (Ω**)** ≤ Average time (Θ) ≤ Upper Bound (O)

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.

For insertion sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *0* | *1* | *2* | *3* | *4* | *5* | *6* |
| 30 | 50 | 57 | 78 |  |  |  |

The worst case occurs when the array is reverse sorted. For example, when we add the number **10** into the array, all elements of the array move backward in one position.

The best case occurs when the array is sorted in the same order as output. For example, when we add the number **90** into the array without any change.

**III. Asymptotic Notations**

**1. Θ Notation**

Lightbox

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.

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}

**2. O Notation:**

BigO

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

O(g(n)) = {f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ c\*g(n) for all n ≥ n0}

For example, consider the case of Insertion Sort. It takes quadratic time in worst case. We can say that the time complexity of Insertion sort is O(n2).

**3. Ω Notation**

BigOmega

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω(g(n)) = {f(n): there exist positive constants c and n0

such that 0 ≤ c\*g(n) ≤ f(n) for all n ≥ n0}

**IV. Properties of Asymptotic Notations:**

Some important properties of those notations

**1. General properties:**

If f(n) is O(g(n)) then k\*f(n) is also O(g(n)); where k is a constant.

Example: f(n) = 3n2 + 5 is O(n2)

Then we have 7\*f(n) = 7(3n2 + 5) = 21n2 + 35 is also O(n2)

Similarly, this property satisfies for 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) = 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.

**3. Reflexive properties:**

If f(n) is given, then f(n) is O(f(n)). Since max value of f(n) will be f(n) itself.

Example: f(n) = n2; O(n2) that is O(f(n))

Similarly, this property satisfies for both Θ and Ω notation.

**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.