8.TYPES OF ANALYSIS

It is illogical to assume that the efficiency of algorithm depends solely on the input size **n** . Sometimes , efficiency of an algorithm depends on the distribution of input data as well.

Worst Case , Best Case and Average Case efficiency of algorithms can be estimated by considering different distributions of input data.

- Let D_n be the set of all valid instances of a given problem.
- Here, n is the input size.
- Let there be k different instances, $k \in D_n$.
- The input size of k is also n.
- Let $T_i(n)$ be the complexity of the algorithm for the input instance i , where i ranges from 1 to k .

WORST CASE COMPLEXITY

AND

UPPER BOUND

WORST CASE COMPLEXITY

DEFINITION: The *worst-case efficiency* of an algorithm is its efficiency for the worst-case input of size n , which is an input (or inputs) of size n, for which the algorithm runs the longest among all possible inputs of that size.

The worst-case complexity defines $T(n)$ as a complexity function $W(n)$ of the worst-case input for which the algorithm:

- Maximum Time.
- Needs to perform the maximum number of steps or operations to process the input for accomplishing the given task.

Mathematically, the worst-case complexity can be defined as:

$$W(n) = max_{1 \le i \le k}\{T_i(k)\}$$

The worst-case complexity is a pessimistic analysis assuming the worst scenario. It gives a performance guarantee that in the worst-case algorithm would take $W(n)$.

UPPER BOUND

The worst-case complexity of an algorithm gives an *upper bound* of the algorithm.

The upper bound of an algorithm is the measure of the maximum computational effort required to solve a given problem.

Let's consider a problem of linear search for a list of numbers given in the Table.

Index	1	2	3	....	100
Element	23	34	12	....	100

Formally a stated a linear search problem is given an array of numbers and a target key. The worst-case complexity of linear search occurs when the item is either present as the last item or not present in the list at all.

Such bound which constitutes all the elements and algorithm traverses almost all the element to get the target is known as *Upper Bound*. And such upper bound is covered by the worst-case efficiency.

For example, let us consider a scenario of searching for a number 170. To decide whether this is the last element of an array or to infer that it is not present in the array, n comparisons are required. Therefore, the worst-case complexity of this algorithm can be given as $W(n) = n$.

BEST CASE COMPLEXITY AND LOWER BOUND

BEST CASE COMPLEXITY

DEFINITION: The best-case efficiency of an algorithm is its efficiency for the best-case of size n, which is an input(or inputs) of size n for which the algorithm runs fastest among all possible inputs of that size.

The best case analysis is given as the minimum computation time of the algorithm for all that instances of the *Domain*.

Recollect that a *Domain is the set of all valid inputs of an algorithm*.

It can be stated as follows :

$$B(n) = min_{1 \le i \le k}\{T_i(k)\}$$

The Best Case for Linear Search is 1 , if the element found at first index.

$$B(n) = 1$$

LOWER BOUND OF AN ALGORITHM:

In real time , best cases are rare in real world problems and rarely reflect the efficiency of an algorithm . It represents the ideal or lower bound of an algorithm. This means that the algorithm requires a minimum amount of effort.

Lower Bound are such bound that constitutes all the element and algorithm needs to traverse least amount of elements to get the target is known as Lower Bound.

LOWER BOUND VS UPPER BOUND

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either.

- We should not expect to get best-case inputs, we might be able to take advantage of the fact that for some algorithms a good best-case performance extends to some useful types of inputs close to being the best-case ones.

That is If a problem has k possible solutions , the lower bound is given by the best algorithm considering all k possible solutions.

FOR EXAMPLE: *"There is a sorting algorithm for which the best-case inputs are already sorted arrays on which the algorithm works very fast."*

Moreover, the best-case efficiency deteriorates only slightly for almost-sorted arrays. Therefore, such an algorithm might well be the method of choice for applications dealing with almost – sorted arrays.

And, of-course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis. Hence, it is often difficult to compute the lower bound of an algorithm.

AVERAGE-CASE COMPLEXITY

The average-case analysis assumes that the input is random and provides prediction about the running time of an algorithm for a random input.

For a linear search problem, the average case complexity can be given as follows:

$$A(n) = \sum_{i=1}^{k} P_i \times T(n)$$

Here P_i is the probability with the respect to the instance I . The average cost measures the average amount of resources the program consumes, assuming that the inputs of the algorithm are random.

The average-case complexity is neither the best nor the worst case. It indicates the average performance of an algorithm. For a linear search problem average-case complexity can be calculated as follows.

- The item can be present anywhere in the list.

Thus, the probability of the presence of the item in the given array is equally likely:

$$p = \frac{1}{n}$$

Hence the item can be present anywhere in the array.

Therefore, the number of comparisons required is as follows:

$$= \left(1 \times \frac{1}{n}\right) + \left(2 \times \frac{1}{n}\right) + \left(3 \times \frac{1}{n}\right) + \cdots + \left(n \times \frac{1}{n}\right)$$

$$= \frac{1}{n}(1 + 2 + 3 + \cdots + n)$$

$$= \frac{1}{n} \times \frac{n(n+1)}{2}$$

$$= \frac{n+1}{2}$$

$$\approx \frac{n}{2}$$