Computational Complexity

An algorithm is a well-defined list of steps for solving a particular problem. It is a sequence of computational steps that transform the input into the desired output.

The complexity of an algorithm is the function which gives the running time and/or space in terms of input size. The performance of a program is measured through its complexity.

Computational complexity focuses on the amount of computing resources needed for a particular algorithm to run efficiently. This efficiency of a program can be measured in terms of space and time.

Space Complexity: The space complexity of an algorithm is the amount of memory required for its execution.

Time Complexity: The time complexity of an algorithm is the amount of time required for its execution.

Estimating Complexity of an Algorithm

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm.

Actual runtime of an algorithm depends upon speed of computer, amount of RAM, operating system and the compiler being used. Hence we cannot use actual runtime of algorithms for comparison between two algorithms.

So, for time complexity estimation of an algorithm, we need to count the number of elementary instructions that executes this algorithm. The number is computed with respect to the size n of the input data.

Big-O notation

It is a method of representing the upper bound of algorithm's running time. It is also known as Omega Notation. Using Big-O notation, we can give longest amount to time taken by the algorithm to complete.

e.g. if we have two algorithms having time complexity O(N) and O(N^2) respectively, then by varying value of N, we can easily check which algorithm is more efficient, i.e. which algorithm takes less time to run.

Classification of Different Order of Growth

Class Name	Order of Growth	Description	Example
Constant	1	As input size grows, we get larger running time	Scanning array elements
Logarithmic	log n	The algorithm does not consider all inputs. It is divided into smaller parts in each integration	Performing binary search
Linear	n	Running time depends on the input size n	Sequential search
n log n	n log n	Some instance of inputs are considered in a list of size n	Merge sort/ Quick sort
Quadratic	n^2	When algorithm has two nested loops	Scanning Matrix elements
Cubic	n^3	When algorithm has three nested loops	Matrix multiplication
Exponential	2^n	When algorithm has very faster rate of growth	

Calculating Complexity

For finding out the time complexity of a piece of code, we need to consider for loops, nested loops, if-then-else, consecutive statements and logarithmic complexity.

(i) for loop

```
for(i=0;  i<N;  i++)
{
        count ++;
}
```

Every time this statement will take constant time i.e. 'C' for execution.
Loop will execute for i values from 0 to n-1.

Total time = C*N i.e. O(N) → Linear

(ii) for nested loop

```
for(i=0;  i<N;  i++)
{
        for(j=0;  j<N;  j++)
        {
                count ++;
        }
}
```

The outer loop will be executed N times and for every iteration of outer loop, the inner loop will be executed N times. So the inner loop will execute N x N times.

Total time = C*N*N times = CN^2, i.e. O(N^2) → Quadratic

(iii) if-then-else statement along with a single for loop

```
if(amount>cost + tax)                    //constant time c0
{
  count ++;                              // c1 time
  while(count < n)                       // loop will execute n times i.e. n(c2+c3)
  {
        amount = cost + tax              // c2 time
        count ++;                        // c3 time
  }
  Printf("Capacity = %d",count);         //c4 time
}
else
{
        printf ("Insufficient fund");    // c5 time
}
```

Total time = $C_0 + C_1 + n(C_2+C_3) + C_4 + C_5$, i.e. O(n).

(iv) consecutive statements with conditions and loops

```
for(i=0;  i<n;  i++)
{
        A[i] = (1-t) * X[i] + t * Y[i];      // constant time C0
        B[i] = (1-s) * X[i] + s * Y[i];      // constant time C1
}
for(i=0;  i<N;  i++)
{
        for(j=0;  j<N;  j++)
```

```
        {
            C[i][j] = j * A[i] + B[j];          // constant time C2
        }
    }
```

Total Time = $n*(C_0+C_1) + n^2*C_2$ i.e. $O(n^2)$

(v) logarithmic complexity

Algorithm taking logarithmic time are commonly found in operations on binary trees or when using binary search. An algorithm is said to take logarithmic time if,

Total time T(n) = O(log n)

Best, Average and Worst Case Complexity

There are three cases to analyze algorithmic complexities.

(i) **Best Case Complexity** – Here we calculate lower bound of running time of an algorithm. We must know the case that causes **minimum number of operations to be executed**. It describes an algorithm's behavior under optimal conditions.

(ii) **Worst Case Complexity** – Here we calculate upper bound of running time of an algorithm. We must know the case that causes maximum number of operations to be executed.

(iii) **Average Case Complexity** – Here we consider all possible inputs and calculate computing time for all the inputs.

Analyzing Algorithms

(i) **Bubble Sort**: In Best, Worst and Average case, Bubble sort takes $O(n^2)$ time

(ii) **Selection Sort**: In Best, Worst and Average case, Bubble sort takes $O(n^2)$ time

(iii) **Insertion Sort**: Best Case→When the array is already sorted, it takes O(n) time
Worst Case→ In worst case, insertion sort takes $O(n^2)$ time

(iv) **Linear Search**: A linear search is a sequential search. A linear search sequentially moves through your collection looping for a matching value.
Best Case → here the value must be present in the first position of the list. The complexity is O(1) i.e. constant
Worst Case → this is an unsuccessful search. The complexity is O(n) time

(v) **Binary Search**: Best Case → Binary search gives the best case when the element to be searched is the middle element. Best Case complexity of Binary search is O(1), i.e. constant
Worst Case and Average Case, the complexity is O(log n) time.