Unit 1: Complexity Analysis

Data Structure

Data structure is the way of storing data in a computer so that it can be used efficiently.

There are two types of data structure

1. **Linear Data Structure**: When the elements are stored on contiguous memory locations then data structure is called linear data structure. For example, array, stack, queue etc.
2. **Non Linear Data Structure**: In nonlinear data structure, elements are stored in non-contiguous memory locations. Eg tree, graphs, etc.

Data structure can be static or dynamic in nature.

A static data structure is one whose capacity is fixed at creation. An array is an example of static data structure.

A dynamic data structure is one whose capacity is variable, so it can expand or contract at any time. Linked List, binary tree are example of dynamic data structure.

Operations on Data Structure

- Traversing
- Searching
- Sorting
- Insertion
- Deletion

Algorithm: An algorithm is finite set of instructions to perform the computational task, is finite number of steps. To develop a program of an algorithm, we select an appropriate data structure for that algorithm. Therefore algorithm and its associated data structures form a program.

Algorithm + Data Structure = Program

Data structures are building blocks of a program.

Properties of Algorithm

- **Input**: The quantity that is given to algorithm initially is called input.
- **Output:** The quantity produced by algorithm is called output. The output will have some relationship with input.
- **Finiteness**: The algorithm must terminate after finite number of steps.
- **Definiteness**: Each step of the algorithm must be precisely defined that is each step should be unambiguous.

Different means of expressing algorithms

- Natural Language
- Pseudo code
- Flowchart
- Programming Language

Complexity Analysis of Algorithm

Several algorithms could be created to solve a single problem. These algorithms may vary in the way they get, process and output data. They could have significant differences in terms of performance and space utilization.

It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time/space requirements as a function input size.

Time Complexity: The time complexity of an algorithm measures the amount of time taken by an algorithm to run as a function of input.

Space Complexity: The space complexity of an algorithm measures the amount of space taken by an algorithm to run as function of input.

Types of Analysis

- **Worst Case Running Time:** The worst case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. For expressing worst case running time of an algorithm Big O notation is used.
- **Best Case Running Time:** The best case running time of an algorithm is lower bound on the running time for any input. The best rarely occurs in practice. For expressing best case running time of an algorithm Big Ω notation is used.
- **Average Case Running Time:** The average case running time of an algorithm is an estimate of the running time for an "average input". For expressing average case running time of an algorithm Big Θ notation is used.
- **Amortized Analysis:** In amortized analysis, the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis is concerned with the overall cost of arbitrary sequences. It is the average performance of each operation in the worst case. It guarantees the average performance of each operation in the worst case.
  Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive.

Big O(oh) Notation:

The big O notation gives the asymptotic upper bounds of the running time of an algorithm.

A function f(n) is O(g(n)) if there exists two positive constants c and N such that f(n)<=c*g(n) for all n>N. We say that g(n) is asymptotic upper bound for f(n).

Properties of Big O Notation: One of the widely used algorithm analysis is a big oh notation. When analyzing algorithm using big O, there are few properties that will help to determine the upper bound of the running time of algorithms.

- **Property 1: Coefficient**
  if f(n) is c*g(n) then f(n) is O(g(n)).
- **Property 2: Sum**
  If f1(n) is O(g(n)) and f2(n) is O(g(n)) then f1(n)+f2(n) is O(g(n)). This property is useful when an algorithm contains several loops of the same order.
- **Property 3 Sum**
  If f1(n) is O(g1(n)) and f2(n) is O(g2(n)) then f1(n)+f2(n) is O(max(g1(n), g2(n)). This algorithm works because we are only concerned with the term of highest growth rate.
- **Property 4: Product**
  If f1(n) is O(g1(n)) and f2(n) is O(g2(n)) then f1(n)*f2(n) is O(g1(n)*g2(n)). This property is useful for analyzing segments of an algorithm with the nested loops.
  If f1(n) is $O(n^2)$ and f2(n) is O(n) then $O(n^2)$ * O(n) which is $O(n^3)$.

Big Omega Notation:

A function f(n) is Ω(g(n)) if there exist positive constants c and N such that for all n≥0, 0≤cg(n)≤f(n). That is, if n is big enough larger then cg(n) will be smaller than f(n). For example, 5x2+6 is Ω(g(n)) because $0 \leq 20n^2 \leq 5n2+6$ whenever n≥4 with constants c=20 and n=4.

Big Theta Notation:

A function f (n) is Θ (g (n)) is there exist positive constants c_1 and c_2 and N such that, for all n≥N, $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$. That is, if n is big enough larger than N, then $c_1 g(n)$ will be smaller than f(n) and $c_2 g(n)$ will be larger than f(n). For example, $5x^2+6$ is Θ (n^2) because $n^2 < 5n^2+6 < 6n^2$ whenever n>5 and $c_1=1$ and $c_2 = 6$.

Possible Problems

All the notations serve the purpose of comparing the efficiency of various algorithms designed for solving the same problem. However, if only big Os are used to represent the efficiency of algorithms, then some of them may be rejected prematurely. The problem is that in the definition of big O notation, f is considered O(g(n)) if the inequality $f(n) \le cg(n)$ holds in the long run for all natural numbers except very few exceptions. This is enough to meet the conditions of the definition. However, this may be of little practical significance if the constant c in $f(n) \le cg(n)$ is prohibitively large, say 10^8.

Consider that there are two algorithms to solve a certain problem and suppose that the number of operations required is $10^8 n$ and $10n^2$. The first function is O(n) and the second is O(n^2). Using just the big O information, the second algorithm is rejected because the number of steps grows too fast. It is true but, again in the long run, because for $n \le 10^7$, which is 10 million, the second algorithm performs fewer operations than the first. In this case the second algorithm is preferable.

For these reasons, it may be desirable to use one more notation that includes constants which are very large for practical reasons.

Examples of Complexities

There are different Big- O expressions such as O(1), O(log n), O(n), O(nlogn), O(n^2), O(n^3) and O(2^n) and their commonly used names are:

- **O(1): Constant time**. This means an increase in the amount of data size (n) as no effect.
- **O(log n)**: **Logarithmic time**. This means when operations increase once each time n doubles.
- **O(n)**: **Linear time**. The linear time complexity means operation time also increases with the order of n.
- **O(nlogn): Linear Logarithmic Time**: In linear logarithmic time operation increases in the order of n*logn.
- **O(n^2)**: **Quadratic:** Quadratic Complexity means operation increases with square of input.
- **O(n^3)** : **Cubic complexity**:
- **O(2^n)**: **Exponential complexity**.

The time taken that is number of steps when problem size increase can be summarized in the following table.

Input size (n)	O(1)	O(logn)	O(n)	O(n*logn)	O(n^2)	O(n^3)	O(2^n)
1	1	0	1	0	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4096	65536
32	1	5	32	160	1024	32768	4294967296

The best time in the above list is obviously constant time, and the worst is exponential time which, as we have seen, quickly overwhelms even the fastest computers even for relatively small n. Polynomial growth (linear, quadratic, cubic etc) is considered manageable as compared to exponential growth.

We can say that

O(1)<O(logn)<O(n)<O(n*logn)<O(n^2)<O(n^3)<O(2^n)

Finding Asymptotic Complexity: Example

Asymptotic bounds are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed.

In most cases, we are interested in time complexity, which usually measures the number of assignments and comparisons performed during the execution of a program

Let us consider the following program.

Example 1

```
for(i=0,sum=0;i<n;i++)
sum = sum+a[i];
```

First, two variables are initialized, then the for loop iterates n times, and during each iteration, it executes two assignments, one of which updates sum and the other of which updates i. Thus, there are 2+2*n assignments for the complete run of this for loop; its asymptotic complexity is O(n).

Example 2

```
for(i=0;i<n;i++){
        for(j=1,sum = a[0];j<=i;j++){
        Sum+ = a[j];
        System.out.println ("Sum for subarray 0 through "+i+" is "+sum);
}
}
```

Before the loops start, i is initialized. The outer loop is performed n times, executing in each iteration an inner for loop, print statement, and assignment statements for i, j and sum. The inner loop is executed i times for each iϵ{1,……n-1} with two assignments in each iteration; one for sum and one for j. Therefore, there are $1+3n+\Sigma 2i = 1+3n+2(1+2+3+4+……+n-1) = 1+3n+n(n-1) = n^2+2n+1 = O(n^2)$

Algorithms with nested loops usually have a large complexity than algorithms with one loop, but it does not have to grow at all. For example, we may request printing sums of numbers in the last five cells of the subarrays starting in position 0. We adopt the foregoing code and transform it to

```
for(i=4;i<n;i++){
        for(j=i-3,sum = a[i-4];j<=I;j++)
        sum+ = a[j];
       System.out.println ("sum for subarray "+(i-4)+" through  "++"  is "+sum);
}
```

The outer loop is executed n-4 times. For each i, the inner loop is executed only four times; For each iteration of the outer loop, there are eight assignments in the inner loop, and this number does not depend on the size of the array. With the initialization of i, n-4 auto increments of i, and n-4 initializations of j and sum, the program makes 1+8.(n-4) = O(n) assignments.

Analysis of these two examples is relatively uncomplicated because the number of times the loops executed did not depend on the ordering of the arrays.

```
int binarySearch(int [] , int key){
int lo = 0, mid, hi = arr.length-1;
while(low<=hi){
mid = (lo+hi)/2;
if(key<arr[mid])
hi = mid-1;
else if (arr[mid]<key)
```

```
lo = mid+1;
else
return mid;
}
return -1;
}
```

Let us consider the size of an array be n. If key is in the middle of the array, the loop executes only one time. Otherwise, the algorithm looks at one of the halves of size n/2, then at one of the halves of this half, of size $n/2^2$ and so on, until the array is of size 1. Hence, we have the sequence n/2, $n/2^2$,…….$n/2^m$ and we want to know the value of m. But the last term of this sequence $n/2^m$ equals 1, from which we have m = logn. So the fact that k is not in the array can be determined after logn iterations of the loop.

Computational complexity

Computational complexity is a branch of computer science and mathematics that deals with analysis of algorithms. It deals with nature of algorithms and classifies according to their complexity.

Complexity Classes

The analysis of algorithms and the big O() notations allow us to talk about the efficiency of a particular algorithm. However, they have nothing to say about whether there could be a better algorithm for the problem at hand. The field of **complexity analysis** analyzes problems rather than algorithms. The first gross division is between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time, no matter what algorithm is used.

There are several complexity classes in the theory of computation. Some major classes are as follows:

1. Class P:

The complexity class p is the set of decision problems that can be solved by a deterministic algorithm in polynomial time. Problems belonging to p are said to have efficient algorithms. Any algorithm having complexity of lower order polynomial is accepted as an efficient algorithm.
The class of problems which can be solved in time $O(n^k)$ for some k is called class P problem. These problems are sometimes called **easy problems,** because the class contains those problems with running times like O(logn) and O(n). But is also contains those with time O (n^{100}), so the name “easy” should be taken too literally.
Example: The problem of sorting **n** numbers can be done in O($\mathbf{n^2}$) time using quick sort algorithm in worse case. Thus all sorting problems are in P.

Decision Problem:

A problem that has only two answers “yes” and “no” is called decision problem. For example, the question “Is the number N prime?

Deterministic Algorithm:

A deterministic algorithm is a uniquely defined (determined) sequence of steps for a particular input. That is, given an input and a step during execution of the algorithm, there is only one way to determine the next step that the algorithm can make.

Non deterministic Algorithm:

A non-deterministic algorithm is an algorithm that can use a special operation that makes a guess when a decision is to made.

2. Class NP:

The class of decision problems that have verification algorithms with polynomials complexity is known as complexity class NP.

The notation NP actually refers non-deterministic polynomial time algorithms

Example 1:

Closed Tour: Given n cities and integer k, is there a tour, of length less than k, of the cities which begin and ends at the same city?

Example 2:

Chromatic Number (Color): Given a graph and an integer k, is there is way to color the vertices with 'k' colors such that adjacent vertices are colored differently?

The P=NP question: The question of whether NP is the same set as P that is whether the problem that can be solved in non-deterministic polynomial time can be solved in deterministic time is one of the most important open question in theoretical computer science. Due to the wide implication a solution would present. If it were true, many important problems would be shown to have efficient solutions. The P=NP is one of the millennium prize problems proposed by the **Clay Mathematics Institute**. The solution of which is a USD 1 million prize for the first person to provide a solution.

Problem Reduction: A problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q.

NP Hard:

A problem is NP hard if all the problems in NP can be polynomially reduced to it.

An example of an NP hard problem is the optimization problem of finding the least cost cyclic route through all nodes of the weighted graph. This is commonly known as the Travelling Sales man Problem (TSP).

NP Complete: The NP complete are the hardest problems among the NP class. The NP complete is set of decision problems X such that

1 X ϵ NP

2 Every problem in NP is reducible to X i.e. NP-complete are the problems among the NP class

- NP complete problems are the most difficult problem in NP

Example 1: The Boolean satisfiability problem is NP complete.
Boolean satisfiability problem: Given a Boolean expression E, decide if there is some assignment to the variable in E such that E is true:
E = (x1 V x2 V x3) ^ (x4 V x5 V x6)
It is the first problem to be proved to be NP complete by Cook

Example 2: The Hamiltonian cycle problem: Given a graph 'G', does it have a Hamiltonian cycle?

Hamiltonian Cycle: A Hamiltonian cycle of an undirected graph is simple cycle that visits each and every vertex exactly once.