Algorithms

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/ Unambiguous**: Each step of the algorithm must be precisely defined that is each step should be unambiguous.
- **Effectiveness**: Each step in an algorithm should be effective.

Different means of expressing algorithms

- Natural Language
- Pseudo code
- Flowchart
- Programming Language

Algorithm to test whether given number is odd or even:

Step 1: Start
Step 2: Read an number, say x.
Step 3: Find modulus of x by 2 (i.e. a = x%2)
Step 4: if a = 0
Print x is even
else
Print x is odd
Step 5: Stop

Data Structure

Data structure is the way of storing data in a computer so that it can be used efficiently. There are following 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. E.g., 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

Abstract Data Type (ADT):

- An **abstract data type** is a basic mathematical concept that defines the data type.
- ADT consists of set of data values and associated operations that are precisely specified independent of any particular implementation.
- In defining an ADT as a mathematical concept, we are not concerned with space and time efficiency. Those are implementation issues.
- The definition of an ADT is not concerned with implementation details at all.
- ADT is a useful tool for specifying the logical properties of a data type.
- ADT may be implemented using a particular data structure.
- Access is only allowed through that interface.
- Abstract data types are often called user-defined data types, because they allow programmers to define new types that resemble primitive data types.
- The examples of abstract data types are Stacks, Queues, Lists, Trees, Sets

Example1: If we take a SET as an abstract data type, then

- {1, 2, 3, 4}, {5, 3, 6} is the data values.
- Abstract data type Set has the operations EmptySet(S), Insert(x, S), Delete(x, S), Intersection(S1,S2), Union(S1,S2), Member(x, S), Equal(S1,S2), Subset(S1,S2).
- The data values of ADT can be represented as arrays or lists which is the **data structure.**
- The implementation of operations can be different depending upon different programming language which is called **algorithm.**

Example 2: A stack of element of type T is a finite sequence of elements of T together with the operation:

1. Initialize the stack to be empty
2. Determine if the stack is empty or not.
3. Determine if the stack is full or not.
4. If the stack is not full, then insert a new node at one end of the stack, called its **top**.
5. If the stack is not empty, then delete the node at its **top**.

Advantages of Data Structure:

- Data Structure allows information storage on hard disks.
- Data Structure provides means for management of large dataset such as internet or databases.
- Data Structure are necessary for the design of efficient algorithms.
- Data Structure allows safe storage of information on a computer which are available for future use.
- Data Structure allows easier processing of data.

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.

Computational Complexity:

It indicates how much effort is needed to apply an algorithm or how costly it is.

Two Criteria:

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

2. **Space Complexity:** The space complexity of an algorithm measures the amount of space taken by an algorithm to run as function of input.
   - The factor of time is usually more important than that of space.
   - Run time is always system-dependent.
   - Run time also depends on the language in which a given algorithm is written

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.

To evaluate an algorithm's efficiency, the size *n* of a file or an array and the amount of time *t* required to process the data should be used.

Asymptotic Complexity:

- Any terms that do not substantially change the function's magnitude should be eliminated from the function
- The resulting function gives only an approximate measure of efficiency of the original function
- However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data.
- This measure of efficiency is called *asymptotic complexity*
- It is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found.

Example

$f(n) = n^2 + 100n + \log_{10} n + 1{,}000$

n	*f*(*n*)	n^2		100*n*		$\log_{10} n$		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

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 and only if there exists two positive numbers 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).

e.g.: $f(n) = 2n^2 + 3n + 1 =$ ***$O(n^2)$***

Properties of Big O Notation:

- **Property 1 (Coefficient)**:
  If f(n) is ***c*g(n)***, then f(n) is ***O(g(n))***.

- **Property 2 (Sum)**:
  If f(n) is ***O(h(n))*** and g(n) is ***O(h(n))***, then f(n)+g(n) is ***O(h(n))***.

- **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))***.

- **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))***.
  - If f1(n) is ***O(n^2)*** and f2(n) is ***O(n)***, then O(n^2)* O(n) which is ***O(n^3)***.
- **Property 5 (Transitivity)**:
  If f(n) is ***O(g(n))*** and g(n) is ***O(h(n))***, then f(n) is ***O(h(n))***.

Big Omega Notation: The big ***Ω*** notation gives the asymptotic lower bounds of the running time of an algorithm.

A function f(n) is ***Ω(g(n))*** if there exists two positive numbers c and N such that **f(n) ≥ cg(n)** for all n ≥ N.
The only difference between this definition and the definition of big-O notation is the direction of the inequality (i.e. >= and <=).

There is an interconnection between these two notations expressed by the equivalence

f (*n*) is ***Ω(g(n))*** iff ***g(n) is O(f (n))***

Big Theta Notation:

A function f(n) is ***Θ(g(n))*** is there exist positive numbers c_1, c_2 and N such that
$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n ≥ N.

We see that:

f (*n*) is ***Θ(g(n))*** if *f*(*n*) is ***O(g(n))*** and *f*(*n*) is ***Ω(g(n))***.

For example, $5x^2+6$ is $\Theta(n^2)$ because $n^2<5n^2+6<6n^2$ whenever n>5 and $c_1=1$ and $c_2 = 6$.

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: **$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$**

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).