Overview of Data

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Below is an overview of some popular linear data structures.

1. Array
2. Linked List
3. Stack
4. Queue

Array
Array is a data structure used to store homogeneous elements at contiguous locations. Size of an array must be provided before storing data.

Example: For example, let us say, we want to store marks of all students in a class, we can use an array to store them. This helps in reducing the use of number of variables as we don’t need to create a separate variable for marks of every subject. All marks can be accessed by simply traversing the array.

Linked List
A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.

Types of Linked List:
1. Singly Linked List: In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. Doubly Linked List: In this type of Linked list, there are two references associated with each node, One of the reference points to the next node and one to the previous node. Advantage of this data structure is that we can traverse in both the directions and for deletion we don’t need to have explicit access to previous node. Eg. NULL<->1<->2<->3<->NULL
3. Circular Linked List: Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Advantage of this data structure is that any node can be made as starting node. This is useful in implementation of circular queue in linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]

Example: Consider the previous example where we made an array of marks of student. Now if a new subject is added in the course, its marks also to be added in the array of marks. But the size of the array was fixed and it is already full so it can not add any new element. If we make an array of a size lot more than the number of subjects it is possible that most of the array will remain empty. We reduce the space wastage Linked List is formed which adds a node only when a new element is introduced. Insertions and deletions also become easier with linked list.

Stack
A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of
push and pop takes place at the same end that is top of the stack. It can be implemented by using both array and linked list.

**Example:** Stacks are used for maintaining function calls (the last called function must finish execution first), we can always remove recursion with the help of stacks. Stacks are also used in cases where we have to reverse a word, check for balanced parenthesis and in editors where the word you typed the last is the first to be removed when you use undo operation. Similarly, to implement back functionality in web browsers.

**Queue**

A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection. (The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list. **Example:** Queue as the name says is the data structure built according to the queues of bus stop or train where the person who is standing in the front of the queue (standing for the longest time) is the first one to get the ticket. So any situation where resources are shared among multiple users and served on first come first server basis. Examples include CPU scheduling, Disk Scheduling. Another application of queue is when data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

**Circular Queue** The advantage of this data structure is that it reduces wastage of space in case of array implementation, As the insertion of the (n+1)’th element is done at the 0’th index if it is empty.

**Algorithms**

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.
The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** – The maximum number of steps taken on any instance of size \( a \).
- **Best-case** – The minimum number of steps taken on any instance of size \( a \).
- **Average case** – An average number of steps taken on any instance of size \( a \).

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare bubble sort and merge sort. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.
Time Complexity

It’s a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

Space Complexity

It’s a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by \( T(n) \), where \( n \) is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- \( O \) – Big Oh
- \( \Omega \) – Big omega
- \( \theta \) – Big theta
- \( o \) – Little Oh
- \( \omega \) – Little omega

O: Asymptotic Upper Bound

‘O’ (Big Oh) is the most commonly used notation. A function \( f(n) \) can be represented is the order of \( g(n) \) that is \( O(g(n)) \), if there exists a value of positive integer \( n \) as \( n_0 \) and a positive constant \( c \) such that –

\[ f(n) \leq c \cdot g(n) \]

for \( n > n_0 \) in all case

Hence, function \( g(n) \) is an upper bound for function \( f(n) \), as \( g(n) \) grows faster than \( f(n) \).
Example

Let us consider a given function, \( f(n) = 4n^3 + 10n^2 + 5n + 1 \)

Considering \( g(n) = 3g(n) = n^3 \)

\( f(n) \leq 5g(n) \) \( f(n) \leq 5g(n) \) for all the values of \( n > 2 \)

Hence, the complexity of \( f(n) \) can be represented as \( O(g(n))O(g(n)) \), i.e. \( O(n^3)O(n^3) \)

\( \Omega \): Asymptotic Lower Bound

We say that \( f(n) = \Omega(g(n))f(n) = \Omega(g(n)) \) when there exists a constant \( c \) that \( f(n) \geq c \cdot g(n) \) \( f(n) \geq c \cdot g(n) \) for all sufficiently large value of \( n \). Here \( n \) is a positive integer. It means function \( g \) is a lower bound for function \( f \); after a certain value of \( n, f \) will never go below \( g \).

Example

Let us consider a given function, \( f(n) = 4n^3 + 10n^2 + 5n + 1 \)

Considering \( g(n) = 3g(n) = n^3 \), \( f(n) \geq 4g(n) \) \( f(n) \geq 4g(n) \) for all the values of \( n > 0 \)

Hence, the complexity of \( f(n) \) can be represented as \( \Omega(g(n))\Omega(g(n)) \), i.e. \( \Omega(n^3)\Omega(n^3) \)

\( \Theta \): Asymptotic Tight Bound

We say that \( f(n) = \Theta(g(n))f(n) = \Theta(g(n)) \) when there exist constants \( c_1 \) and \( c_2 \) that \( c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \) \( c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \) for all sufficiently large value of \( n \). Here \( n \) is a positive integer.

This means function \( g \) is a tight bound for function \( f \).

Example

Let us consider a given function, \( f(n) = 4n^3 + 10n^2 + 5n + 1 \)

Considering \( g(n) = 3g(n) = n^3 \), \( 4g(n) \leq f(n) \leq 5g(n) \) \( 4g(n) \leq f(n) \leq 5g(n) \) for all the large values of \( n \).

Hence, the complexity of \( f(n) \) can be represented as \( \Theta(g(n))\Theta(g(n)) \), i.e. \( \Theta(n^3)\Theta(n^3) \).

\( O \) - Notation

The asymptotic upper bound provided by \( O \)-notation may or may not be asymptotically tight. The bound \( 2n^2 = O(n^2) \) \( 2n^2 = O(n^2) \) is asymptotically tight, but the bound \( 2n = O(n^2) \) \( 2n = O(n^2) \) is not.

We use \( o \)-notation to denote an upper bound that is not asymptotically tight.

We formally define \( o(g(n)) \) (little-oh of \( g \) of \( n \)) as the set \( f(n) = o(g(n)) \) for any positive constant \( c > 0 \) \( c > 0 \) and there exists a value \( n_0 > 0 \) \( n_0 > 0 \), such that \( 0 \leq f(n) \leq c \cdot g(n) \) \( 0 \leq f(n) \leq c \cdot g(n) \).
Intuitively, in the $o$-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as $n$ approaches infinity; that is,

$$\lim_{n \to \infty} (f(n)g(n)) = 0$$

Example

Let us consider the same function, $f(n) = 4n^3 + 10n^2 + 5n + 1$.

Considering $g(n) = n^4$, we have

$$\lim_{n \to \infty} (4n^3 + 10n^2 + 5n + 1) = 0$$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $o(n^4)$.

$\omega$ – Notation

We use $o$-notation to denote a lower bound that is not asymptotically tight. Formally, however, we define $\omega(g(n))$ (little-omega of $g$ of $n$) as the set $f(n) = \omega(g(n))$ for any positive constant $C > 0$ and there exists a value $n_0 > 0$, such that $0 \leq c \cdot g(n) < f(n) \leq c \cdot g(n)$.

For example, $n^2 = \omega(n)$, but $n^2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that the following limit exists

$$\lim_{n \to \infty} (f(n)g(n)) = \infty$$

That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as $n$ approaches infinity.

Example

Let us consider the same function, $f(n) = 4n^3 + 10n^2 + 5n + 1$.

Considering $g(n) = n^2$, we have

$$\lim_{n \to \infty} (4n^3 + 10n^2 + 5n + 1) = \infty$$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))o(g(n))$, i.e. $o(n^2)o(n^2)$. 