UNIT I – INTRODUCTION

Data structures - Abstract data types - Primitive data structures- – Performance analysis – Space complexity – Time complexity – Asymptotic notations – Performance measurement – Array as an abstract data type – Polynomial as an abstract data type – Sparse matrix abstract data type – String abstract data type.

Data:

A collection of facts, concepts, figures, observations, occurrences or instructions in a formalized manner.

Information:

The meaning that is currently assigned to data by means of the conventions applied to those data(i.e. processed data)

Record:

Collection of related fields.

Data type:

Set of elements that share a common set of properties used to solve a program.

Data Structures:

Data Structure is the way of organizing, storing, and retrieving data and their relationship with each other.

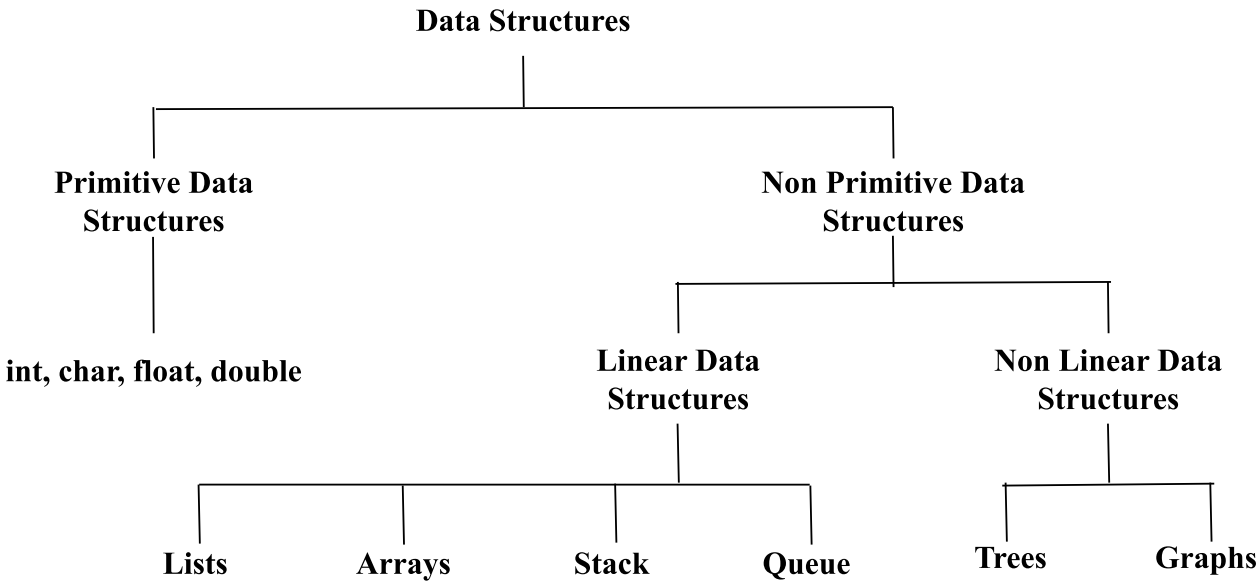
Characteristics of data structures:

1. It depicts the logical representation of data in computer memory.
2. It represents the logical relationship between the various data elements
3. It helps in efficient manipulation of stored data elements.
4. It allows the programs to process the data in an efficient manner.

Operations on Data Structures:

1. Traversal
2. Search
3. Insertion
4. Deletion

Classification of Data Structures



Primary Data Structures/Primitive Data Structures:

Primitive data structures include all the fundamental data structures that can be directly manipulated by machine-level instructions. Some of the common primitive data structures include integer, character, real, boolean etc

Secondary Data Structures/Non Primitive Data Structures:

Non-primitive data structures, refer to all those data structures that are derived from one or more primitive data structures. The objective of creating non-primitive data structures is to form sets of homogeneous or heterogeneous data elements.

Linear Data Structures:

Linear data structures are data structures in which all the data elements are arranged in i, linear or sequential fashion. Examples of data structures include arrays, stacks, queues, linked lists, etc.

Non Linear Data Structures:

In nonlinear data structures, there is a definite order or sequence in which data elements are arranged. For instance, non-linear data structures could arrange data elements in a hierarchical fashion. Examples of non-linear data structures are trees and graphs.

Static and dynamic data structure:

Static Ds: If a ds is created using static memory allocation, ie. ds formed when the number of data items are known in advance ,it is known as static data static ds or fixed size ds.

Dynamic Ds: If the ds is created using dynamic memory allocation i.e ds formed when the number of data items are not known in advance is known as dynamic ds or variable size ds.

Application of data structures:

- Data structures are widely applied in the following areas:
- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

ABSTRACT DATA TYPES (ADTS):

An abstract Data type (ADT) is defined as a mathematical model with a collection of operations defined on that model. Set of integers, together with the operations of union, intersection and set difference form an example of an ADT. An ADT consists of data together with functions that operate on that data.

Advantages/Benefits of ADT:

1. Modularity
2. Reuse
3. Code is easier to understand
4. Implementation of ADTs can be changed without requiring changes to the program that uses the ADTs.

TIME AND SPACE COMPLEXITY OF ALGORITHM

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

Time Complexity

The time complexity of an algorithm quantifies the amount of time taken by an

algorithm to run as a function of the length of the input. The time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running. In order to calculate time complexity on an algorithm, it is assumed that a constant time c is taken to execute one operation, and then the total operations for an input length on N are calculated. Consider an example to understand the process of calculation: Suppose a problem is to find whether a pair (X, Y) exists in an array *A* of *N* elements whose sum is *z*. The simplest idea is to consider every pair and check if it satisfies the given condition or not.

The pseudo-code is as follows:

```
int a[n];
for(int i = 0;i < n;i++)
        cin >> a[i];
for(int i = 0;i < n;i++)
        for(int j = 0;j < n;j++)
                if(i!=j && a[i]+a[j] == z)
                        return true
return false
```

Assume that each of the operations in the computer takes approximately constant time c. The number of lines of code executed actually depends on the value of *z*. During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals z. In the worst case,

- *N**c operations are required for input.
- The outer loop i, runs *N* times.
- For each i, the inner loop j loop runs *n* times.

So total execution time is ***N**c + *N**N**c + c**. Now ignore the lower order terms since the lower order terms are relatively insignificant for large input, therefore only the highest order term is taken (without constant) which is ***N**N*** in this case. Different notations are used to describe the limiting behavior of a function, but since the worst case is taken so big-O notation will be used to represent the time complexity.

Hence, the time complexity is $\mathbf{O(N^2)}$ for the above algorithm. Note that the time complexity is based on the number of elements in array A i.e the input length, so if the length of the array will increase the time of execution will also increase.

Order of growth is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease.

Another Example: Let's calculate the time complexity of the below algorithm:

```
count = 0
for(int i =N;  i > 0; i /= 2)
        for (int j = 0; j < i; j++)
                count++;
```

It seems like the complexity is **O(N * log N)**. **N** for the **j′s** loop and **log(N)** for **i′s** loop. But it's wrong. Let's see why.

Think about how many times **count++** will run.

- When i = N, it will run N times.
- When i = N / 2, it will run N / 2 times.
- When i = N / 4, it will run N / 4 times.
- And so on.

The total number of times **count++** will run is **N + N/2 + N/4+…+1= 2 * N**. So the time complexity will be **O(N)**. Some general time complexities are listed below with the input range for which they are accepted in competitive programming:

Input Length	**Worst Accepted Time Complexity**	**Usually type of solutions**
10 -12	O(N!)	Recursion and backtracking
15-18	O(2N * N)	Recursion, backtracking, and bit manipulation
18-22	O(2N * N)	Recursion, backtracking, and bit manipulation
30-40	O(2N/2 * N)	Meet in the middle, Divide and Conquer
100	O(N4)	Dynamic programming, Constructive
400	O(N3)	Dynamic programming, Constructive
2K	O(N2* log N)	Dynamic programming, Binary Search, Sorting, Divide and Conquer

10K	O(N2)	Dynamic programming, Graph, Trees,
1M	O(N* log N)	Sorting, Binary Search, Divide and Conquer
100M	O(N), O(log N), O(1)	Constructive, Mathematical, Greedy Algorithms

Space Complexity

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

The pseudo-code is as follows:

```
int freq[n];
int a[n];
for(int i = 0; i<n; i++)
{
        cin>>a[i];
        freq[a[i]]++;
}
```

Here two arrays of length N, and variable i are used in the algorithm so, the total space used is N * c + N * c + 1 * c = 2N * c + c, where c is a unit space taken. For many inputs, constant c is insignificant, and it can be said that the space complexity is O(N).

There is also auxiliary space, which is different from space complexity. The main difference is where space complexity quantifies the total space used by the algorithm, auxiliary space quantifies the extra space that is used in the algorithm apart from the given input. In the above example, the auxiliary space is the space used by the freq[] array because that is not part of the given input. So total auxiliary space is N * c + c which is O(N) only.

ASYMPTOTIC ANALYSIS – AVERAGE AND WORST-CASE ANALYSIS

Worst Case Analysis (Usually Done)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be

executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of linear search would be Θ(n).

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Average Case Time $= \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$

$= \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$

$= \Theta(n)$

Best Case Analysis (Bogus)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant(not dependent on n). So time complexity in the best case would be

(1). Most of the time, we do worst-case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm. The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, Algorithms may take years to run. For some algorithms, all the cases are asymptotically the same, i.e., there are no worst and best cases. For example, Merge Sort.

Merge Sort does Θ(nLogn) operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick

Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the array into two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

ASYMPTOTIC NOTATION

The main idea of asymptotic analysis is to measure the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

1. Θ notation
2. Big O notation
3. Ω notation

Θ Notation:

The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants.

For example, consider the following expression. $\mathbf{3n^3 + 6n^2 + 6000 = \Theta(n^3)}$

Dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

Big O Notation

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

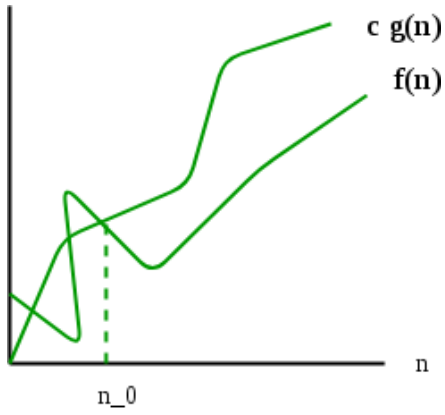
If we use Θ notation to represent time complexity of Insertion sort, we have to use

two statements for best and worst cases:

1. The worst-case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= f(n) <= c*g(n) for all n >= n0}



f(n) = O(g(n))

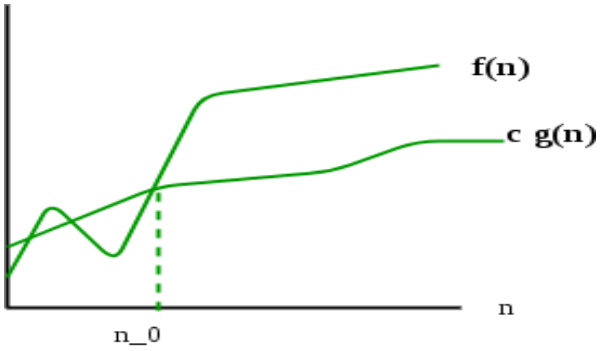
Ω Notation

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation can be useful when we have a lower bound on the time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst- case and sometimes in the average case.



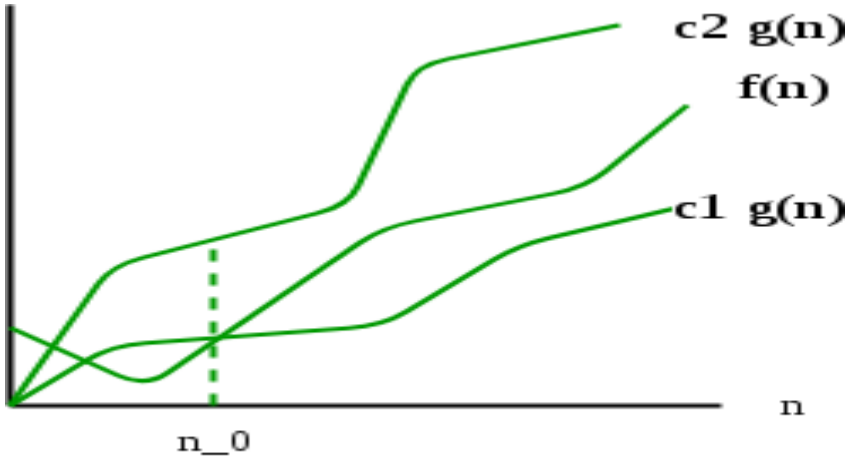
f(n) = Omega(g(n))

Properties of Asymptotic Notations

As we have gone through the definition of these three notations let's now discuss some important properties of those notations.

1. General Properties:

If f(n) is O(g(n)) then a*f(n) is also O(g(n)), where a is a constant.



f(n) = theta(g(n))

Example: f(n) = $2n^2+5$ is $O(n^2)$

then 7*f(n) = $7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

2. Transitive Properties:

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if f(n) = n, g(n) = n^2 and h(n)=n^3

n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

3. Reflexive Properties:

Reflexive properties are always easy to understand after transitive.
If f(n) is given then f(n) is O(f(n)). Since MAXIMUM VALUE OF f(n) will be f(n) itself! Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.
Example: $f(n) = n^2$; $O(n^2)$ i.e O(f(n))
Similarly, this property satisfies both Θ and Ω notation.
We can say that:
If f(n) is given then f(n) is Θ(f(n)). If f(n) is given then f(n) is Ω (f(n)).
If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)) ; where a is a constant. If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)) ; where a is a constant.

4. Symmetric Properties:

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .
Example: $f(n) = n^2$ and $g(n) = n^2$ then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$
This property only satisfies for Θ notation.

5. Transpose Symmetric Properties:

If f(n) is O(g(n)) then g(n) is Ω (f(n)).
Example: $f(n) = n$, $g(n) = n^2$ then n is $O(n^2)$ and n^2 is Ω (n)
This property only satisfies O and Ω notations.

PROGRAM PERFORMANCE MEASUREMENT

Performance measurement and program evaluation can both help identify areas of programs that need improvement and determine whether the program is achieving its goals or objectives. They serve different but complementary functions:

- Performance measurement is an ongoing process that monitors and reports on a program's progress and accomplishments by using pre-selected performance measures.
- Program evaluation, however, uses measurement and analysis to answer specific questions about how well a program is achieving its outcomes and why.

What is Program Evaluation?

Program evaluations are individual systematic studies conducted to assess how well a program is working and why. EPA has used program evaluation to:

- Support new and innovative approaches and emerging practices

- Identify opportunities to improve efficiency and effectiveness
- Continuously improve existing programs
- Subsequently, improve human health and the environment

What Types of Program Evaluations are there?

Program Evaluation:-

Program evaluations can assess the performance of a program at all stages of a program's development. The type of program evaluation conducted aligns with the program's maturity (e.g., developmental, implementation, or completion) and is driven by the purpose for conducting the evaluation and the questions that it seeks to answer. The purpose of the program evaluation determines which type of evaluation is needed.

Design Evaluation:-

A design evaluation is conducted early in the planning stages or implementation of a program. It helps to define the scope of a program or project and to identify appropriate goals and objectives. Design evaluations can also be used to pre-test ideas and strategies.

Process Evaluation:-

A process evaluation assesses whether a program or process is implemented as designed or operating as intended and identifies opportunities for improvement. Process evaluations often begin with an analysis of how a program currently operates. Process evaluations may also assess whether program activities and outputs conform to statutory and regulatory requirements, EPA policies, program design or customer expectations.

Outcome Evaluations:-

Outcome evaluations examine the results of a program (intended or unintended) to determine the reasons why there are differences between the outcomes and the program's stated goals and objectives (e.g., why the number and quality of permits issued exceeded or fell short of the established goal?). Outcome evaluations sometimes examine program processes and activities to better understand how outcomes are achieved and how quality and productivity could be improved.

Impact Evaluation:-

An impact evaluation is a subset of an outcome evaluation. It assesses the causal links between program activities and outcomes. This is achieved by comparing the observed outcomes with an estimate of what would have happened if the program had not existed (e.g., would the water be swimmable if the program had not been instituted).

Cost-Effectiveness Evaluation:-

Cost-effectiveness evaluations identify program benefits, outputs or outcomes and compare them with the internal and external costs of the program.

What is performance measurement?

Performance measurement is a way to continuously monitor and report a program's progress and accomplishments, using pre-selected performance measures. By establishing program measures, offices can gauge whether their program is meeting their goals and objectives. Performance measures help programs understand "what" level of performance is achieved.

How do we determine good measures?

Measurement is essential to making cost-effective decisions. We strive to meet three key criteria in our measurement work:

- **Is it meaningful?**
    - Measurement should be consistent and comparable to help sustain learning.
- **Is it credible?**
    - Effective measurement should withstand reasonable scrutiny.
- **Is it practical?**
    - Measurement should be scaled to an agency's needs and budgetary constraints.

How is performance measurement different from program evaluation?

A program sets performance measures as a series of goals to meet over time. Measurement data can be used to identify/flag areas of increasing or decreasing performance that may warrant further investigation or evaluation. Program evaluations assess whether the program is meeting those performance measures but also look at why they are or are not meeting them.

For example, imagine you bought a new car that is supposed to get 30 miles per gallon. But say, you notice that you are only getting 20 miles per gallon. That's a performance measurement. You looked at whether your car was performing where it should be. So what do you do next? You would take it to a mechanic. The mechanic's analysis and recommendations would be the program evaluation because the mechanic would diagnose why the car is not performing as well as it should.