Unit 1: Chapter 1

Introduction to Data Structures and Algorithms

1.0 OBJECTIVE

After studying this unit, you will be able to:

- Discuss the concept of data structure
- Discuss the need for data structures
- Explain the classification of data structures
- Discuss abstract data types
- Discuss various operations on data structures
- Explain algorithm complexity
- Understand the basic concepts and notations of data structures

1.1 INTRODUCTION

The study of data structures helps to understand the basic concepts involved in organizing and storing data as well as the relationship among the data sets. This in turn helps to determine the way information is stored, retrieved and modified in a computer's memory.

1.2 BASIC CONCEPT OF DATA STRUCTURE

Data structure is a branch of computer science. The study of data structure helps you to understand how data is organized and how data flow is managed to increase efficiency of any process or program. Data structure is the structural representation of logical relationship between data elements. This means that a data structure organizes data items based on the relationship between the data elements.

Example:

A house can be identified by the house name, location, number of floors and so on. These structured set of variables depend on each other to identify the exact house. Similarly, data structure is a structured set of variables that are linked to each other, which forms the basic component of a system

1.2.1 Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

1.2.2 Need for Data Structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

1.2.3 Goals of Data Structure

Data structure basically implements two complementary goals.

Correctness: Data structure is designed such that it operates correctly for all kinds of input, which is based on the domain of interest. In other words, correctness forms the primary goal of data structure, which always depends on the specific problems that the data structure is intended to solve.

Efficiency: Data structure also needs to be efficient. It should process the data at high speed without utilizing much of the computer resources such as memory space. In a real time state, the efficiency of a data structure is an important factor that determines the success and failure of the process.

1.2.4 Features of Data Structure

Some of the important features of data structures are:

Robustness: Generally, all computer programmers wish to produce software that generates correct output for every possible input provided to it, as well as execute

efficiently on all hardware platforms. This kind of robust software must be able to manage both valid and invalid inputs.

Adaptability: Developing software projects such as word processors, Web browsers and Internet search engine involves large software systems that work or execute correctly and efficiently for many years. Moreover, software evolves due to ever changing market conditions or due to emerging technologies.

Reusability: Reusability and adaptability go hand-in-hand.
It is a known fact that the programmer requires many resources for developing any software, which makes it an expensive enterprise. However, if the software is developed in a reusable and adaptable way, then it can be implemented in most of the future applications. Thus, by implementing quality data structures, it is possible to develop reusable software, which tends to be cost effective and time saving.

1.3 CLASSIFICATION OF DATA STRUCTURES

A data structure provides a structured set of variables that are associated with each other in different ways. It forms a basis of programming tool that represents the relationship between data elements and helps programmers to process the data easily.

Data structure can be classified into two categories:

1.3.1 Primitive data structure
1.3.2 Non-primitive data structure

Figure 1.1shows the different classifications of data structures.

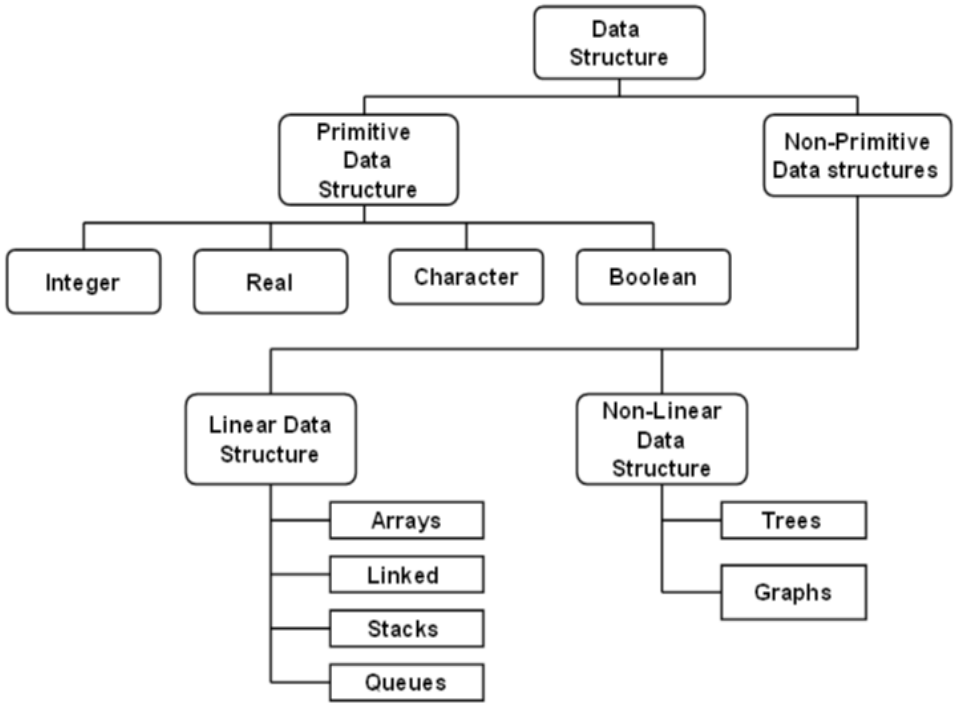


Figure 1.1 Classifications of data structures.

1.3.1 Primitive Data Structure

Primitive data structures consist of the numbers and the characters which are built in programs. These can be manipulated or operated directly by the machine level instructions. Basic data types such as integer, real, character, and Boolean come under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.

1.3.2 Non-primitive Data Structure

Non-primitive data structures are those that are derived from primitive data structures. These data structures cannot be operated or manipulated directly by the machine level instructions. They focus on formation of a set of data elements that is either homogeneous (same data type) or heterogeneous (different data type).
These are further divided into linear and non-linear data structure based on the structure and arrangement of data.

1.3.2.1 Linear Data Structure

A data structure that maintains a linear relationship among its elements is called a linear data structure. Here, the data is arranged in a linear fashion. But in the memory, the arrangement may not be sequential.

Ex: Arrays, linked lists, stacks, queues.

1.3.2.1 Non-linear Data Structure

Non-linear data structure is a kind of data structure in which data elements are not arranged in a sequential order. There is a hierarchical relationship between individual data items. Here, the insertion and deletion of data is not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

I) Array

Array, in general, refers to an orderly arrangement of data elements. Array is a type of data structure that stores data elements in adjacent locations. Array is considered as linear data structure that stores elements of same data types. Hence, it is also called as a linear homogenous data structure.

When we declare an array, we can assign initial values to each of its elements by enclosing the values in braces { }.

int Num [5] = { 26, 7, 67, 50, 66 };

This declaration will create an array as shown below:

	0	1	2	3	4
Num	26	7	67	50	66

Figure 1.2 Array

The number of values inside braces { } should be equal to the number of elements that we declare for the array inside the square brackets [ ]. In the example of array Paul, we have declared 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element. After this declaration, array Paul will have five integers, as we have provided 5 initialization values.

Arrays can be classified as one-dimensional array, two-dimensional array or multidimensional array.

***One-dimensional Array*:** It has only one row of elements. It is stored in ascending storage location.

Two-dimensional Array: It consists of multiple rows and columns of data elements. It is also called as a matrix.

Multidimensional Array: Multidimensional arrays can be defined as array of arrays. Multidimensional arrays are not bounded to two indices or two dimensions. They can include as many indices as required.

Limitations:

- Arrays are of fixed size.

- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

However, these limitations can be solved by using linked lists.

Applications:

- Storing list of data elements belonging to same data type
- Auxiliary storage for other data structures
- Storage of binary tree elements of fixed count
- Storage of matrices

II) Linked List

A linked list is a data structure in which each data element contains a pointer or link to the next element in the list. Through linked list, insertion and deletion of the data element is possible at all places of a linear list. Also in linked list, it is not necessary to have the data elements stored in consecutive locations. It allocates space for each data item in its own block of memory. Thus, a linked list is considered as a chain of data elements or records called nodes. Each node in the list contains information field and a pointer field. The information field contains the actual data and the pointer field contains address of the subsequent nodes in the list.

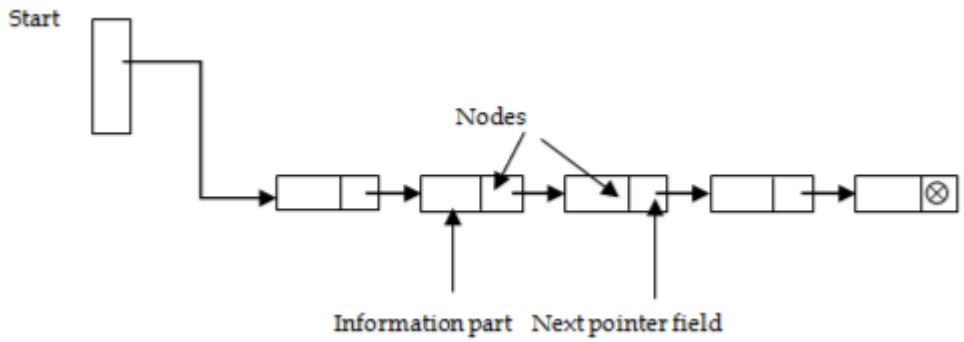


Figure 1.3: A Linked List

Figure 1.3 represents a linked list with 4 nodes. Each node has two parts. The left part in the node represents the information part which contains an entire record of data items and the right part represents the pointer to the next node. The pointer of the last node contains a null pointer.

***Advantage*:** Easier to insert or delete data elements

***Disadvantage*:** Slow search operation and requires more memory space

Applications:

- Implementing stacks, queues, binary trees and graphs of predefined size.
- Implement dynamic memory management functions of operating system.
- Polynomial implementation for mathematical operations
- Circular linked list is used to implement OS or application functions that require round robin execution of tasks.
- Circular linked list is used in a slide show where a user wants to go back to the first slide after last slide is displayed.
- Doubly linked list is used in the implementation of forward and backward buttons in a browser to move backwards and forward in the opened pages of a website.
- Circular queue is used to maintain the playing sequence of multiple players in a game.

III) Stacks

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

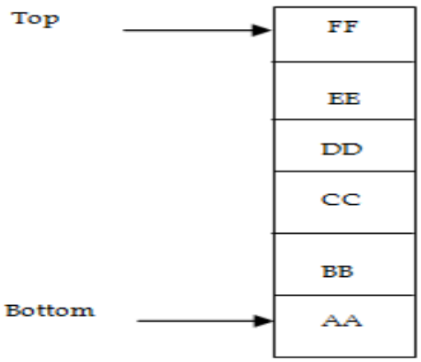


Figure 1.4: A Stack

In the computer's memory, stacks can be implemented using arrays or linked lists. Figure 1.4 is a schematic diagram of a stack. Here, element FF is the top of the stack and element AA is the bottom of the stack. Elements are added to the stack from the top. Since it follows LIFO pattern, EE cannot be deleted before FF is deleted, and similarly DD cannot be deleted before EE is deleted and so on.

Applications:

- Temporary storage structure for recursive operations

- Auxiliary storage structure for nested operations, function calls, deferred/postponed functions
- Manage function calls
- Evaluation of arithmetic expressions in various programming languages
- Conversion of infix expressions into postfix expressions
- Checking syntax of expressions in a programming environment
- Matching of parenthesis
- String reversal
- In all the problems solutions based on backtracking.
- Used in depth first search in graph and tree traversal.
- Operating System functions
- UNDO and REDO functions in an editor.

IV) Queues

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front. Like stacks, queues can be implemented by using either arrays or linked lists.

Figure 1.5 shows a queue with 4 elements, where 55 is the front element and 65 is the rear element. Elements can be added from the rear and deleted from the front.

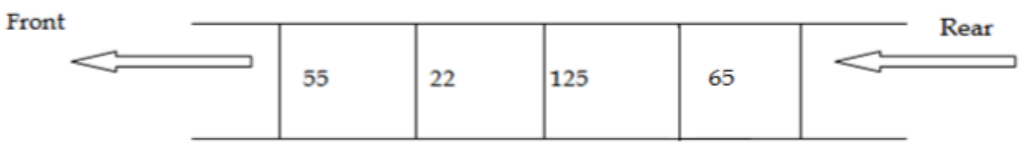


Figure 1.5: A Queue

Applications:

- It is used in breadth search operation in graphs.
- Job scheduler operations of OS like a print buffer queue, keyboard buffer queue to store the keys pressed by users
- Job scheduling, CPU scheduling, Disk Scheduling
- Priority queues are used in file downloading operations in a browser
- Data transfer between peripheral devices and CPU.
- Interrupts generated by the user applications for CPU
- Calls handled by the customers in BPO

V) Trees

A tree is a non-linear data structure in which data is organized in branches. The data elements in tree are arranged in a sorted order. It imposes a hierarchical structure on the data elements.

Figure 1.6 represents a tree which consists of 8 nodes. The root of the tree is the node 60 at the top. Node 29 and 44 are the successors of the node 60. The nodes 6, 4, 12 and 67 are the terminal nodes as they do not have any successors.

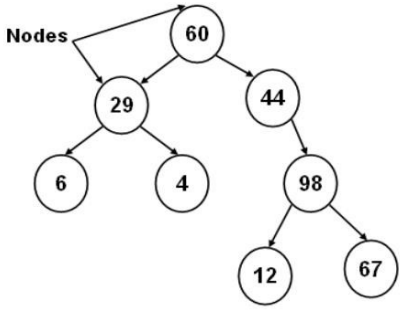


Figure 1.6: A Tree

Advantage: Provides quick search, insert, and delete operations
Disadvantage: Complicated deletion algorithm

Applications:

- Implementing the hierarchical structures in computer systems like directory and file system.
- Implementing the navigation structure of a website.
- Code generation like Huffman's code.
- Decision making in gaming applications.
- Implementation of priority queues for priority-based OS scheduling functions
- Parsing of expressions and statements in programming language compilers
- For storing data keys for DBMS for indexing
- Spanning trees for routing decisions in computer and communications networks
- Hash trees
- path-finding algorithm to implement in AI, robotics and video games applications

VI) Graphs

A graph is also a non-linear data structure. In a tree data structure, all data elements are stored in definite hierarchical structure. In other words, each node has only one parent node. While in graphs, each data element is called a

vertex and is connected to many other vertexes through connections called edges.

Thus, a graph is considered as a mathematical structure, which is composed of a set of vertexes and a set of edges. Figure shows a graph with six nodes A, B, C, D, E, F and seven edges [A, B], [A, C], [A, D], [B, C], [C, F], [D, F] and [D, E].

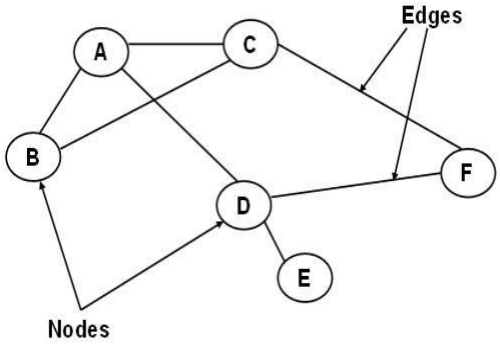


Figure 1.7 Graph

Advantage: Best models real-world situations
Disadvantage: Some algorithms are slow and very complex

Applications:

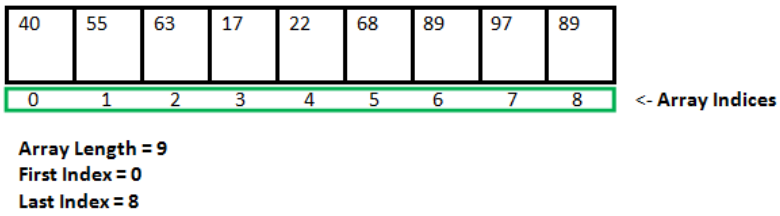
- Representing networks and routes in communication, transportation and travel applications
- Routes in GPS
- Interconnections in social networks and other network-based applications
- Mapping applications
- Ecommerce applications to present user preferences
- Utility networks to identify the problems posed to municipal or local corporations
- Resource utilization and availability in an organization
- Document link map of a website to display connectivity between pages through hyperlinks
- Robotic motion and neural networks

1.4 STATIC DATA STRUCTURE VS DYNAMIC DATA STRUCTURE

Data structure is a way of storing and organising data efficiently such that the required operations on them can be performed be efficient with respect to time as well as memory. Simply, Data Structure are used to reduce complexity (mostly the time complexity) of the code.

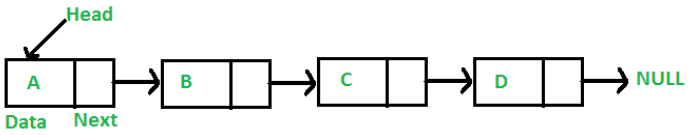
Data structures can be two types:
1. Static Data Structure
2. Dynamic Data Structure

What is a Static Data structure?
In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.



Example of Static Data Structures: **Array**

What is Dynamic Data Structure?
In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.



Example of Dynamic Data Structures: **Linked List**

Static Data Structure vs Dynamic Data Structure
Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code. Static Data Structure provides more easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

1.5 OPERATIONS ON DATA STRUCTURES

This section discusses the different operations that can be performed on the various data structures previously mentioned.

Traversing It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

Inserting It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

Deleting It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

Sorting Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

Merging Lists of two sorted data items can be combined to form a single list of sorted data items.

1.6 ABSTRACT DATA TYPE

According to National Institute of Standards and Technology (NIST), a data structure is an organization of information, usually in the memory, for better algorithm efficiency. Data structures include queues, stacks, linked lists, dictionary, and trees. They could also be a conceptual entity, such as the name and address of a person.

From the above definition, it is clear that the operations in data structure involve higher -level abstractions such as, adding or deleting an item from a list, accessing the highest priority item in a list, or searching and sorting an item in a list. When the data structure does such operations, it is called an abstract data type.

It can be defined as a collection of data items together with the operations on the data. The word "abstract" refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves **what** can be done with the data, not **how** has to be done. For ex, in the below figure the user would be involved in checking that what can be done with the data collected not how it has to be done.

An implementation of ADT consists of storage structures to store the data items and algorithms for basic operation. All the data structures i.e. array, linked list, stack, queue etc are examples of ADT.

Advantage of using ADTs

In the real world, programs *evolve* as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency. In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

1.7 ALGORITHM

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

1.7.1 Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

An algorithm should have the following characteristics −

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

1.7.2 Advantages and Disadvantages of Algorithm

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

1.7.3 Different approach to design an algorithm

1. **Top-Down Approach:** A top-down approach starts with identifying major components of system or program decomposing them into their lower level components & iterating until desired level of module complexity is achieved . In this we start with topmost module & incrementally add modules that is calls.

2. **Bottom-Up Approach:** A bottom-up approach starts with designing most basic or primitive component & proceeds to higher level components. Starting from very bottom , operations that provide layer of abstraction are implemented

1.7.4 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c**
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c**
Step 7 − STOP

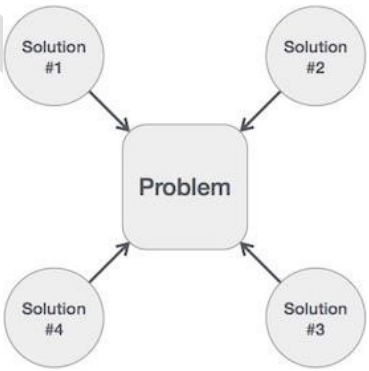
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD
Step 2 − get values of **a** & **b**
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

1.8 ALGORITHM COMPLEXITY

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

1.8.1 Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

```
Algorithm: SUM(A, B)
Step 1 -  START
Step 2 - C ← A + B + 10
Step 3 -  Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

1.8.2 Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

1.9 ALGORITHM ANALYSIS

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- ***A Priori*** **Analysis** or Performance or Asymptotic Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior*** **Analysis** or Performance Measurement − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.

Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.

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. Big-O notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

1.9.1 Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

1.9.1.1 Best Case Analysis

If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best case time complexity. The best case efficiency of an algorithm is the efficiency for the best case input of size **n**. Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.

1.9.1.2 Average Case Analysis

If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.

Average case analysis provides necessary information about an algorithm's behavior on a typical or random input. You must make some assumption about the possible inputs of size **n** to analyze the average case efficiency of algorithm.

1.9.1.3 Worst Case Analysis

If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity. The worst case efficiency of an algorithm is the efficiency for the worst case input of size **n**. The algorithm runs the longest among all the possible inputs of the similar size because of this input of size **n**.

1.10 MATHEMATICAL NOTATION

Algorithms are widely used in various areas of study. We can solve different problems using the same algorithm. Therefore, all algorithms must follow a standard. The mathematical notations use symbols or symbolic expressions, which have a precise semantic meaning.

1.10.1 Asymptotic Notations

A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More accurately, you must be able to judge the time taken to run two solutions, and choose the better among the two.
To select the best algorithm, it is necessary to check the efficiency of each algorithm. The efficiency of each algorithm can be checked by computing its time complexity. The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as the fastest possible, slowest possible or average possible.

The notations such as O (Big-O), Ω (Omega), and θ (Theta) are called as asymptotic notations. These are the mathematical notations that are used in three different cases of time complexity.

1.10.1.1 Big-O Notation

'O' is the representation for Big-O notation. Big -O is the method used to express the upper bound of the running time of an algorithm. It is used to describe the performance or time complexity of the algorithm. Big-O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used by the algorithm.

Table 2.1 gives some names and examples of the common orders used to describe functions. These orders are ranked from top to bottom.

Table 2.1: Common Orders

	Time complexity		Examples
1	O(1)	Constant	Adding to the front of a linked list
2	O(log n)	Logarithmic	Finding an entry in a sorted array
3	O(n)	Linear	Finding an entry in an unsorted array
4	O(n log n)	Linearithmic	Sorting 'n' items by 'divide-and-conquer'
5	O(n2)	Quadratic	Shortest path between two nodes in a graph
6	O(n3)	Cubic	Simultaneous linear equations
7	O(2n)	Exponential	The Towers of Hanoi problem

Big-O notation is generally used to express an ordering property among the functions. This notation helps in calculating the maximum amount of time taken by an algorithm to compute a problem. Big-O is defined as:

$$f(n) \leq c * g(n)$$

where, **n** can be any number of inputs or outputs and **f(n)** as well as ***g(n)*** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer **n0** such that,
$n \geq n0$.

The Big-O can also be denoted as f(n) = O(g(n)), where ***f(n)*** and ***g(n)*** are two non -negative functions and f(n) < g(n) if **g(n)** is multiple of some constant **c**. The graphical representation of f(n) = O(g(n)) is shown in figure 2.1, where the running time increases considerably when **n** increases**.**

Example: Consider f(n)=15n3+40n2+2nlog n+2n. As the value of **n** increases, n3 becomes much larger than n2, nlog n, and n. Hence, it dominates the function f(n) and we can consider the running time to grow by the order of n3. Therefore, it can be written as f(n)=O(n3).

The values of n for f(n) and C* g(n) will not be less than n0. Therefore, the values less than n0 are not considered relevant.

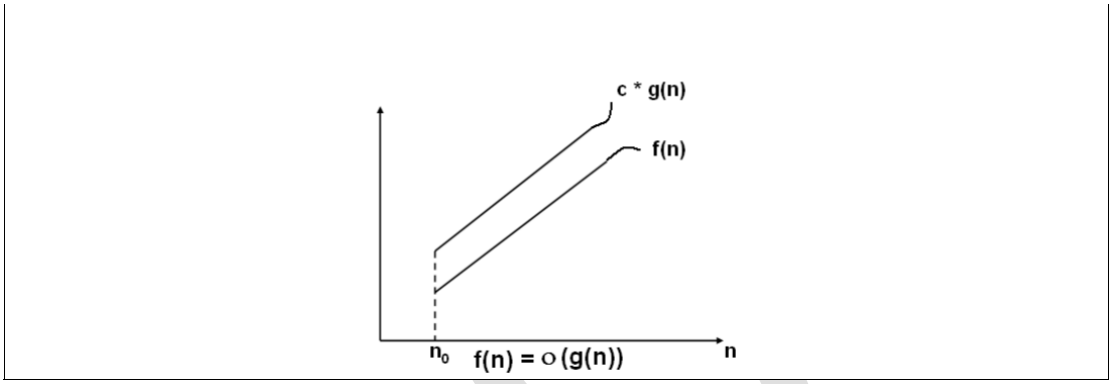


Figure 1.8: Big-O Notation f(n) = O(g(n))

Let us take an example to understand the Big-O notation more clearly.

Example:
Consider function f(n) = 2(n)+2 and g(n) = n2.

We need to find the constant c such that $f(n) \leq c * g(n)$.
Let n = 1, then
f(n) = 2(n)+2 = 2(1)+2 = 4

g(n) = n2 = 12 = 1
Here, f(n)>g(n)
Let n = 2, then
f(n) = 2(n)+2 = 2(2)+2 = 6
g(n) = n2 = 22 = 4

Here, f(n)>g(n)
Let n = 3, then
f(n) = 2(n)+2 = 2(3)+2 = 8
g(n) = n2 = 32 = 9
Here, f(n)<g(n)

Thus, when n is greater than 2, we get f(n)<g(n). In other words, as **n** becomes larger, the running time increases considerably. This concludes that the Big-O helps to determine the 'upper bound' of the algorithm's run-time.

Limitations of Big O Notation

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants. For example, if one algorithm takes O(n2) time to execute and the other takes O(100000n2) time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

1.10.1.2 Omega Notation

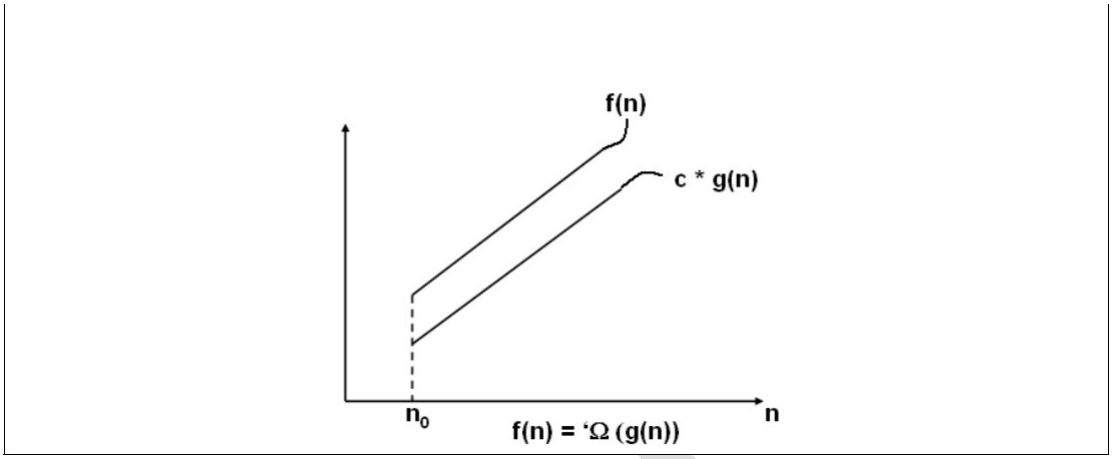
'Ω' is the representation for Omega notation. Omega describes the manner in which an algorithm performs in the best case time complexity. This notation provides the minimum amount of time taken by an algorithm to compute a problem. Thus, it is considered that omega gives the "lower bound" of the algorithm's run-time. Omega is defined as:

$$f(n) \geq c * g(n)$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer **n0** such that n>n0.

Omega can also be denoted as f(n) = Ω (g(n)) where, **f** of **n** is equal to Omega of **g** of **n** . The graphical representation of f(n) = Ω (g(n)) is shown in figure 2.2. The function **f(n**) is said to be in Ω (g(n)), if **f(n)** is bounded below by some constant multiple of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c** and some non-negative integer **n0,** such that f(n) ≥ c ∗ g(n) for all n ≥n0.
Figure 2.2 shows Omega notation.

Figure 1.9 Omega Notation f(n) = Ω (g(n))



Let us take an example to understand the Omega notation more clearly.

Example:
Consider function f(n) = 2n2+5 and g(n) = 7n.
We need to find the constant c such that $f(n) \geq c * g(n)$.

Let n = 0, then

f(n) = 2n2+5 = 2(0)2+5 = 5
g(n) = 7(n) = 7(0) = 0
Here, f(n)>g(n)

Let n = 1, then

f(n) = 2n2+5 = 2(1)2+5 = 7
g(n) = 7(n) = 7(1) = 7
Here, f(n)=g(n)

Let n = 2, then

f(n) = 2n2+5 = 2(2)2+5 = 13

g(n) = 7(n) = 7(2) = 14
Here, f(n)<g(n)

Thus, for n=1, we get $f(n) \geq c * g(n)$. This concludes that Omega helps to determine the "lower bound" of the algorithm's run-time.

1.10.1.3 Theta Notation

'θ' is the representation for Theta notation. Theta notation is used when the upper bound and lower bound of an algorithm are in the same order of magnitude. Theta can be defined as:

$$c1 * g(n) \leq f(n) \leq c2 * g(n) \quad \text{for all } n>n0$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative functions. These functions are true only if there are two constants namely, **c1**, **c2,** and a non-negative integer **n0**.

Theta can also be denoted as f(n) = θ(g(n)) where, **f** of **n** is equal to Theta of **g** of **n**. The graphical representation of f(n) = θ(g(n)) is shown in figure 2.3. The function **f(n)** is said to be in θ (g(n)) if **f(n)** is bounded both above and below by some positive constant multiples of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c1** and **c2** and some non-negative integer **n0,** such that C2g(n)≤f(n)≤ C1g(n) for all n≥n0.

Figure shows Theta notation.

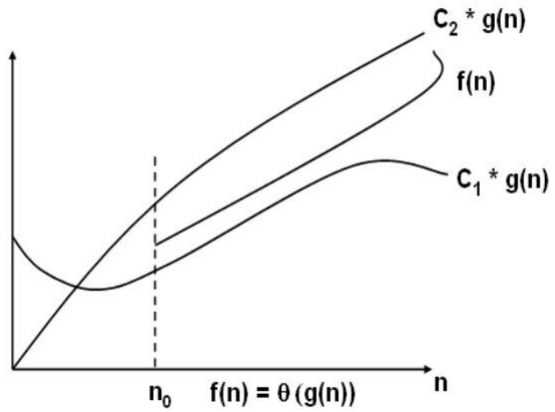


Figure 1.10: Theta Notation f(n) = θ(g(n))

Let us take an example to understand the Theta notation more clearly.

Example: Consider function f(n) = 4n + 3 and g(n) = 4n for all n ≥ 3; and f(n) = 4n + 3 and g(n) = 5n for all n ≥ 3.

Then the result of the function will be:
Let n = 3
f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 4n =4(3) = 12 and

f(n) = 4n + 3 = 4(3)+3 = 15
g(n) = 5n =5(3) = 15 and

here, c1 is 4, c2 is 5 and n0 is 3

Thus, from the above equation we get c1 $g(n)$ $f(n)$ c2 $g(n)$. This concludes that Theta notation depicts the running time between the upper bound and lower bound.

1.11 ALGORITHM DESIGN TECHNIQUE

1.11.1 Divide and Conquer
1.11.2 Back Tracking Method
1.11.3 Dynamic programming

1.11.1 Divide and Conquer

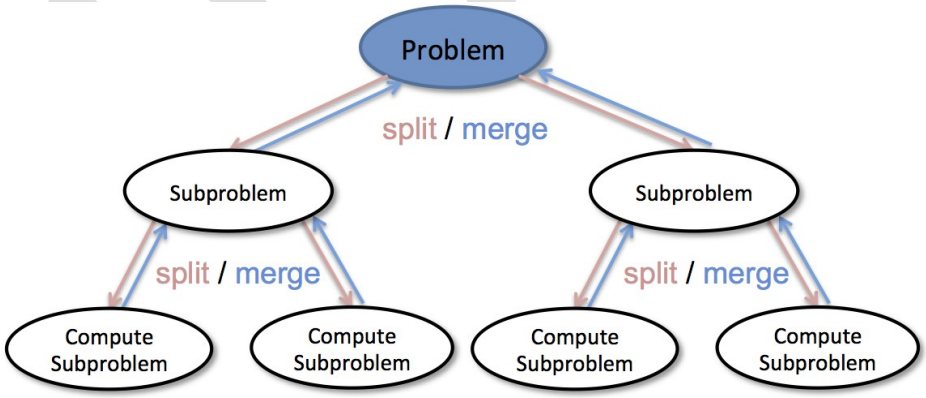
Introduction

Divide and Conquer approach basically works on breaking the problem into sub problems that are similar to the original problem but smaller in size & simpler to solve. once divided sub problems are solved recursively and then combine solutions of sub problems to create a solution to original problem.

At each level of the recursion the divide and conquer approach follows three steps:
Divide: In this step whole problem is divided into several sub problems.

Conquer: The sub problems are conquered by solving them recursively, only if they are small enough to be solved, otherwise step1 is executed.

Combine: In this final step, the solution obtained by the sub problems are combined to create solution to the original problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search

3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamentals of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula, we apply D&C Strategy, i.e., we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So, the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

1.11.2 Backtracking

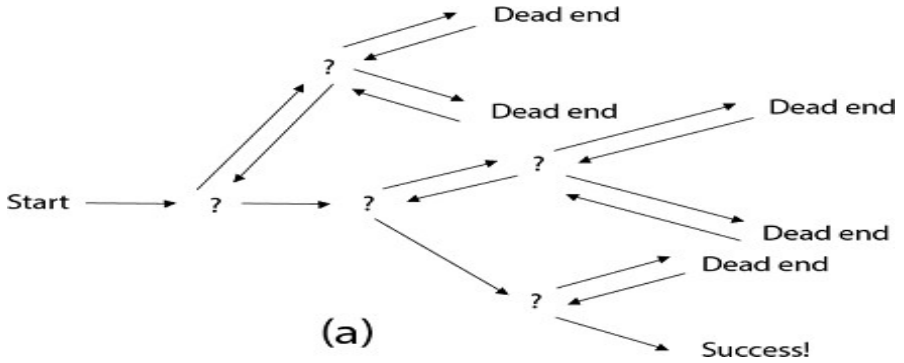
Introduction

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

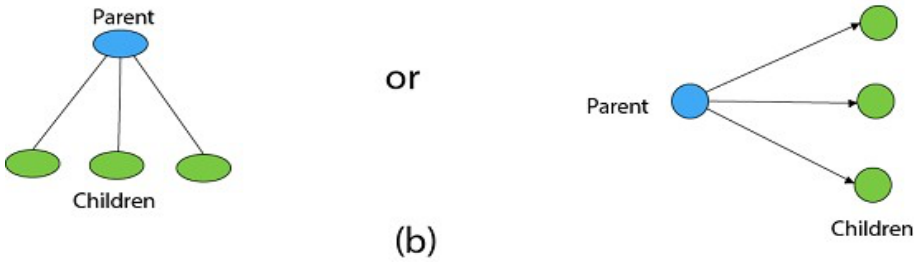
Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

In the following Figure:

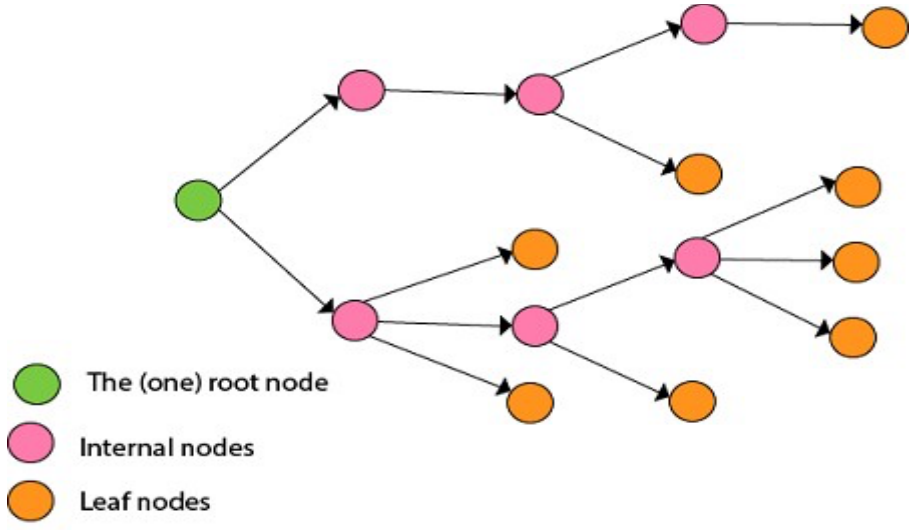
- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Generally, however, we draw our trees downward, with the root at the top.



A tree is composed of nodes.



(c)

Backtracking can understand of as searching a tree for a particular "goal" leaf node.

Backtracking is undoubtedly quite simple - we "explore" each node, as follows:

To "explore" node N:

1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,
   Explore C
   If C was successful, return "success"
4. Return "failure"

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth-first search** with any bounding function. All solution using backtracking is needed to satisfy a complex set of constraints. The constraints may be explicit or implicit.

Explicit Constraint is ruled, which restrict each vector element to be chosen from the given set.

Implicit Constraint is ruled, which determine which each of the tuples in the solution space, actually satisfy the criterion function.

1.11.3 Dynamic programming

Dynamic Programming Technique is similar to divide-and-conquer technique. Both techniques solve a problem by breaking it down into several sub-problems that can be solved recursively. The main difference between is that, Divide & Conquer approach partitions the problems into independent sub-problems, solve the sub-problems recursively, and then combine their solutions to solve the original problems. Whereas dynamic programming is applicable when the sub-problems are not independent, that is, when sub-problems share sub subproblems. Also, A dynamic programming algorithms solves every sub problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub subproblems is encountered.

Therefore **"Dynamic programming is a applicable when sub problem are not independent, that is when sub problem share sub problems."**

As Greedy approach, Dynamic programming is typically applied to optimization problems and for them there can be many possible solutions and the requirement is to find the optimal solution among those. But Dynamic programming approach is little different greedy approach. In greedy solutions are computed by making choices in serial forward way and in this no backtracking & revision of choices is done where as Dynamic programming computes its solution bottom up by producing them from smaller sub problems, and by trying many possibilities and choices before it arrives at the optimal set of choices.

The Development of a dynamic-programming algorithm can be broken into a sequence of four steps:

Divide, Sub problems: The main problems are divided into several smaller sub problems. In this the solution of the main problem is expressed in terms of the solution for the smaller sub problems. Basically, it is all about characterizing the structure of an optimal solution and recursively define the value of an optimal solution.

Table, Storage: The solution for each sub problem is stored in a table, so that it can be used many times whenever required.

Combine, bottom-up Computation: The solution to main problem is obtained by combining the solutions of smaller sub problems. i.e., compute the value of an optimal solution in a bottom-up fashion.

Construct an optimal solution from computed information. **(This step is optional and is required in case if some additional information is required after finding out optimal solution.)**

Now for any problem to be solved through dynamic programming approach it must follow the following conditions:

Principle of Optimality: It states that for solving the master problem optimally, its sub problems should be solved optimally. It should be noted that not all the times each sub problem(s) is solved optimally, so in that case we should go for optimal majority.

Polynomial Breakup: For solving the main problem, the problem is divided into several sub problems and for efficient performance of dynamic programming the total number of sub problems to be solved should be at-most a polynomial number.

Various algorithms which make use of Dynamic programming technique are as follows:

1. Knapsack problem.
2. Chain matrix multiplication.
3. All pair shortest path.
4. Travelling sales man problem.
5. Tower of hanoi.
6. Checker Board.
7. Fibonacci Sequence.
8. Assembly line scheduling.
9. Optimal binary search trees.

1.12 SUMMARY

A data structure is a particular way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently.
There are two types of data structures: primitive and non-primitive data structures.
Primitive data structures are the fundamental data types which are supported by a programming language. Nonprimitive data structures are those data structures which are created using primitive data structures.
Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.
If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. However, if the elements of a data structure are not stored in sequential order, then it is a non-linear data structure.
An array is a collection of similar data elements which are stored in consecutive memory locations.
A linked list is a linear data structure consisting of a group of elements (called nodes) which together represent a sequence.
A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front.

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical tree structure.
A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships can exist between the nodes.
An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.
An algorithm is basically a set of instructions that solve a problem.
The time complexity of an algorithm is basically the running time of the program as a function of the input size.
The space complexity of an algorithm is the amount of computer memory required during the program execution as a function of the input size.
The worst-case running time of an algorithm is an upper bound on the running time for any input.
The average-case running time specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.
The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed and the type of the loop that is being used.

1.13 MODEL QUESTIONS

1. Define data structures. Give some examples.
2. In how many ways can you categorize data structures? Explain each of them.
3. Discuss the applications of data structures.
4. Write a short note on different operations that can be performed on data structures.
5. Write a short note on abstract data type.
6. Explain the different types of data structures. Also discuss their merits and demerits.
7. Define an algorithm. Explain its features with the help of suitable examples.
8. Explain and compare the approaches for designing an algorithm.
9. What do you understand by a graph?
10. Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.
11. What do you understand by time–space trade-off?
12. What do you understand by the efficiency of an algorithm?
13. How will you express the time complexity of a given algorithm?
14. Discuss the significance and limitations of the Big O notation.
15. Discuss the best case, worst case and average case complexity of an algorithm.
16. Categorize algorithms based on their running time complexity.
17. Give examples of functions that are in Big O notation as well as functions that are not in Big O notation.
18. Explain the Ω notation.

19. Give examples of functions that are in Ω notation as well as functions that are not in Ω notation.
20. Explain the Θ notation.
21. Give examples of functions that are in Θ notation as well as functions that are not in Θ notation.
22. Explain the ω notation.
23. Give examples of functions that are in ω notation as well as functions that are in ω notation.

1.14 List of References

https://www.javatpoint.com/
https://www.studytonight.com
https://www.tutorialspoint.com
https://www.geeksforgeeks.org/heap-sort/
https://www.programiz.com/dsa/heap-sort
https://www.2braces.com/data-structures