DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 1

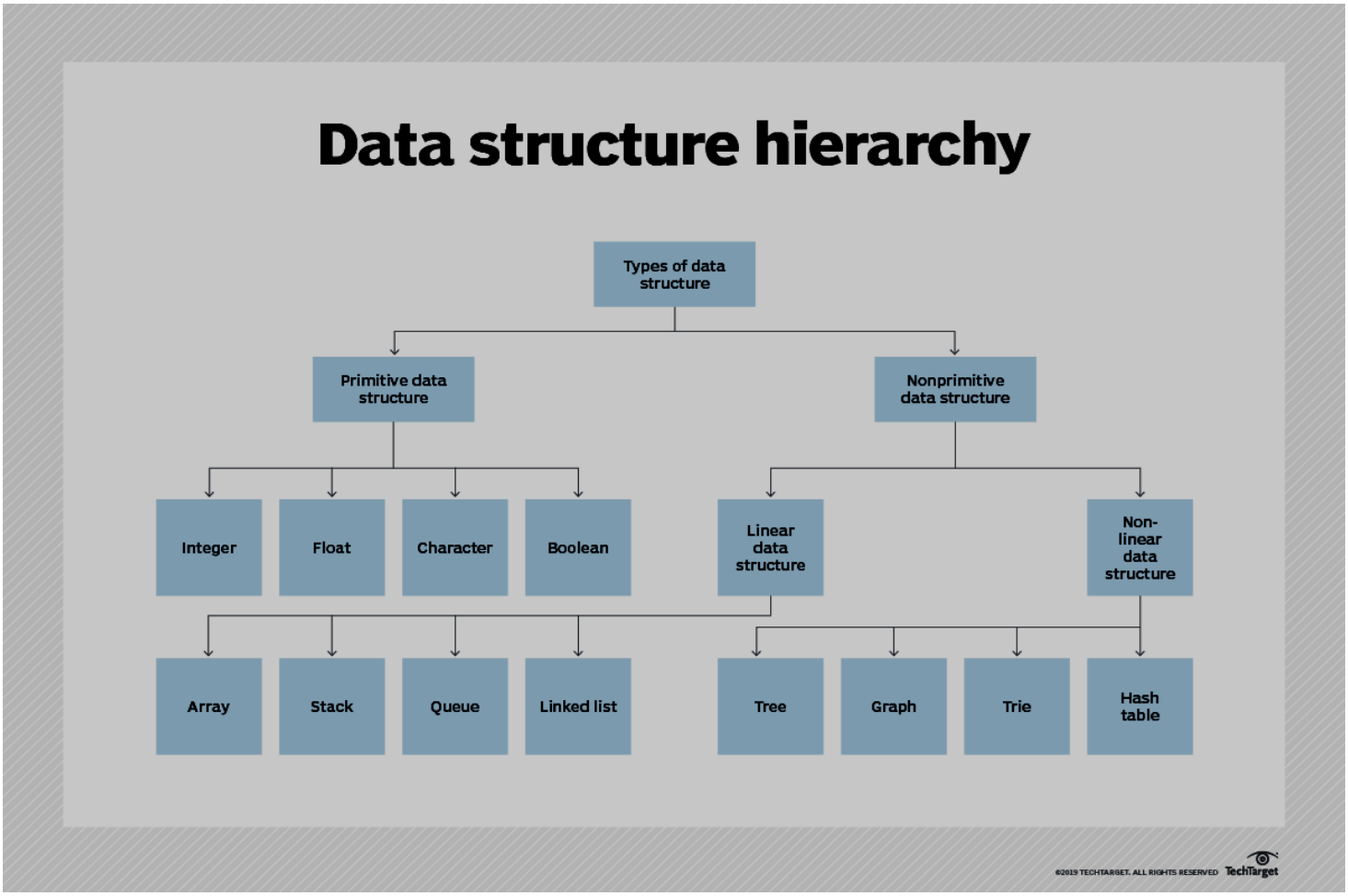
INTRODUCTION TO DATA STRUCTURE: Elementary Data Structures: Stack – Queues – Trees – Priority Queue – Graphs – What is an Algorithm? – Algorithm Specification – Performance Analysis: Space Complexity – Time Complexity – Asymptotic Notation – Randomized Algorithms.

1.1. DATA STRUCTURES

Data Structure:

- A data structure is a specialized format for organizing, processing, retrieving and storing data.
- Data structures make it easy for users to access and work with the data they need in appropriate ways.
- In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms.
- n some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data.
- **Examples of Data Structures:** Arrays, Linked Lists, Stack, Queue, Trees, etc.
- **Category:** It is classified into two categories - **Primitive and Non-Primitive.**
    - ***Non-primitive data structures can be further classified into two categories - Linear and Non-linear.***

The four basic data structure types are linear data structures, tree data structures, hash data structures and graph data structures.



Primitive:

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive:

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -

a. Linear Data Structures
b. Non-Linear Data Structures

- **Linear data structure:**

  - ✓ Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.
  - ✓ *Examples of linear data structures are array, stack, queue, linked list, etc.*

  - ❖ **Static data structure:**
    - ✓ Static data structure has a fixed memory size. It is easier to access theelements in a static data structure.
    - ✓ *An example of this data structure is an array.*
  - ❖ **Dynamic data structure:**
    - ✓ In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space)complexity of the code.
    - ✓ *Examples of this data structure are queue, stack, etc.*

- **Non-linear data structure:**

  - ➢ Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

Characteristics of data structures:

- ➢ Three characteristics are….
- ❖ **Linear or non-linear.** Whether the data items are arranged in sequential order, such as with an array, or in an unordered sequence, such as with a graph.
- ❖ **Homogeneous or heterogeneous.** Describes whether all data items in a given repository are of the same type.
- ❖ **Static or dynamic.** Describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time.

Advantages of Data Structures

- Data structure is a secure way of storing the data on our system.
- Data structures help us to process the data easily.
- Data structures also help us to store the data on the disks very efficiently so that we can easily retrieve the data.

Advantages	Disadvantages
Efficient Data Organization	Complexity
Fast Data Retrieval	Learning Curve
Space Optimization	Memory Overhead
Flexibility and Modularity	Performance Trade-offs

Elementary data types:

- Integer, real, character, Boolean, enumeration, pointer.

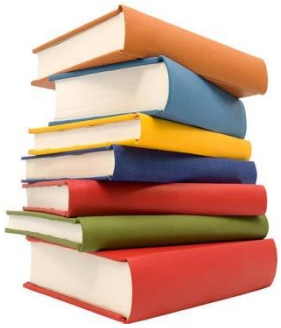
Elementary Data Structures:

- Stacks, Queues, Linked lists, and Root trees.

1.2. STACK

Stack:

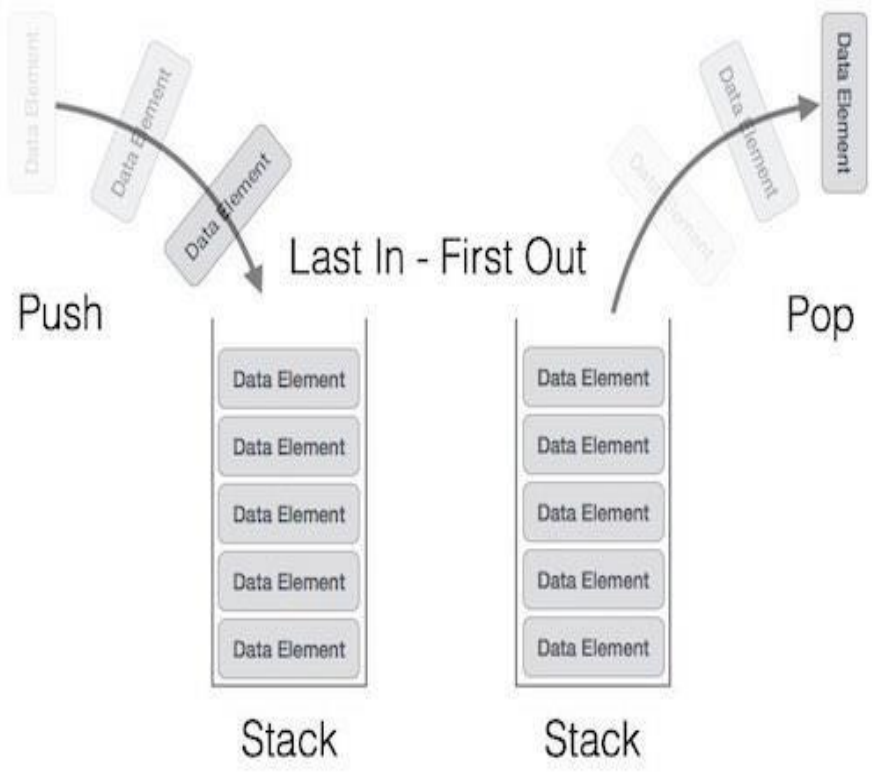
- **Stack is an ordered list** in which all insertions (Push) and deletions (Pop) are made at same end called the top.
- Stack follows **Last in First out (LIFO)** policy so stack is also called as LIFO List.
- A **Stack is a linear data structure** in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.
- Example: Arranging Note / Books in a table, Stack of CD/DVD, Stack of Plates.



Basic Operations on Stacks:

- The most fundamental operations in the stack include: **PUSH(), POP(), STACK_FULL(), STACK_EMPTY().**
- These are all built-in operations to carry out data manipulation and to check the status of the stack.

The following diagram depicts a stack and its operations:



Working of Stack

- ✓ The operations work as follows:
  - A pointer called TOP is used to keep track of the top element in the stack.
  - When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
  - On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
  - On popping an element, we return the element pointed to by TOP and reduce its value.
  - Before pushing, we check if the stack is already full
  - Before popping, we check if the stack is already empty

Types of Stack:

- ✓ There are two types of stacks they are ***Register stack and Memory stack.***
  - **Register stack:** Limited amount of data items are stored in the stack.
  - **Memory stack:** Huge amount of data items are stored in the memory.

Insertion: push()

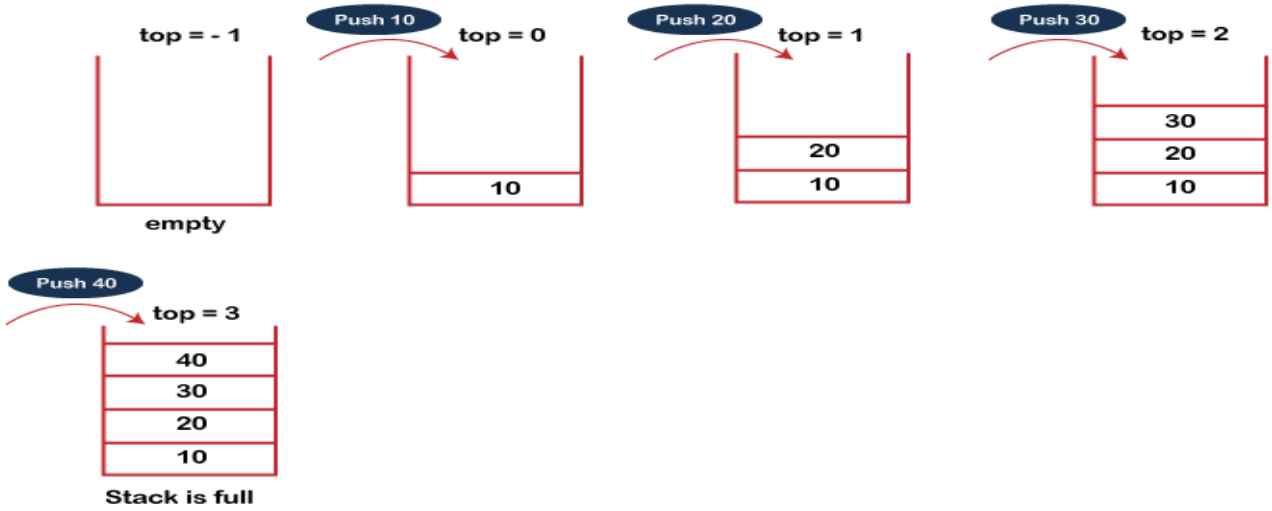
- ➢ push() is an operation that inserts elements into the stack.
- ➢ Before inserting the overflow condition must be checked to make sure that stack not reached the maximum size.
- ➢ The following is an algorithm that describes the push() operation.

```
PUSH(S,x)
1   If STACK_FULL(S)
2       error "overflow"
3   else S.top= S.top+1s
4   S[S.top]=x
```

The above algorithm follow the following steps.

- ✓ Checks if the stack is full.
- ✓ If the stack is full, produces an error and exit.
- ✓ If the stack is not full, increments top to point next empty space.
- ✓ Adds data element to the stack location, where top is pointing.
- ✓ Returns success.

PUSH OPERATION IN STACK



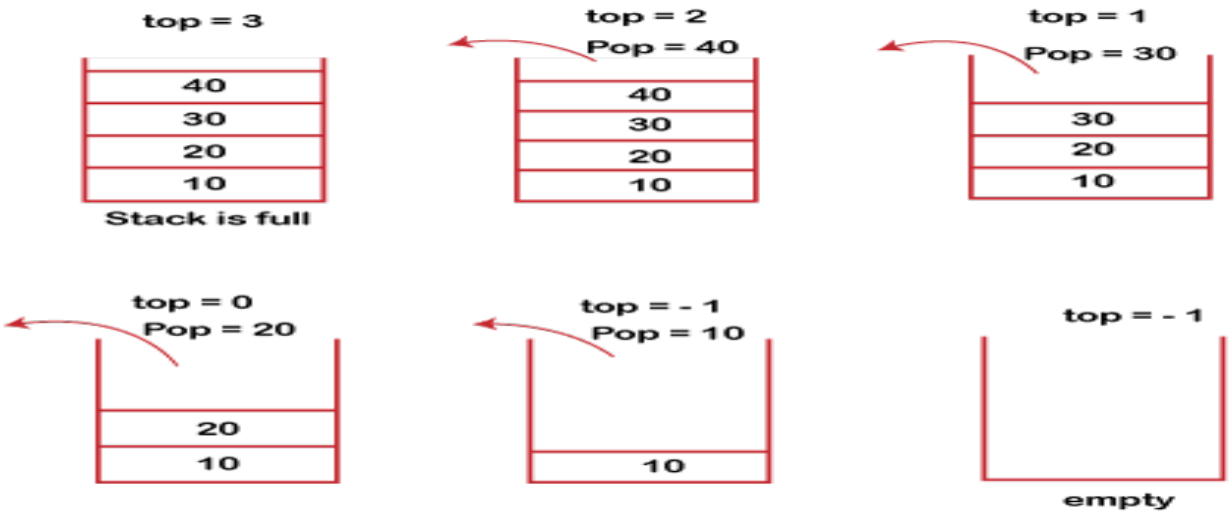
Deletion: pop()

- ➢ pop() is an operation that delete element from the stack.
- ➢ Before deleting the underflow condition must be checked to make sure that stack is not empty.
- ➢ The following is an algorithm that describes the pop() operation.

```
POP(S)
1 if STACK-EMPTY(S)
2   error  "Underflow"
3 else S.top = S.top -1
4 retrun S[S.top +1]
```

The above algorithm follows the following steps.

- ✓ Checks if the stack is empty.
- ✓ If the stack is empty, produces an error and exit.
- ✓ If the stack is not empty, change top to next element position.
- ✓ Returns deleted element.



Applications of STACK:

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**SIASC**" string, so we can achieve this with the help of a stack.
- First, we push all the characters of the string in a stack until we reach the null character.
- After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS (Depth First Search):** This search is implemented on a Graph, and Graph uses the stack datastructure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning ofthe path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
  - Infix to prefix
  - Infix to postfix
  - Prefix to infix
  - Prefix to postfix
  - Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Advantages of Stack:

Advantages	Disadvantages
It is easy to get started	It is not flexible
It does efficient data management	It has a lack of scalability
It has a low hardware Requirement	Unable to Copy & Paste

Anyone with access can edit the program	It has a limited memory size

Real-life examples of a stack:

A deck of cards, piles of books, piles of money, and many more.

Stack Time Complexity:

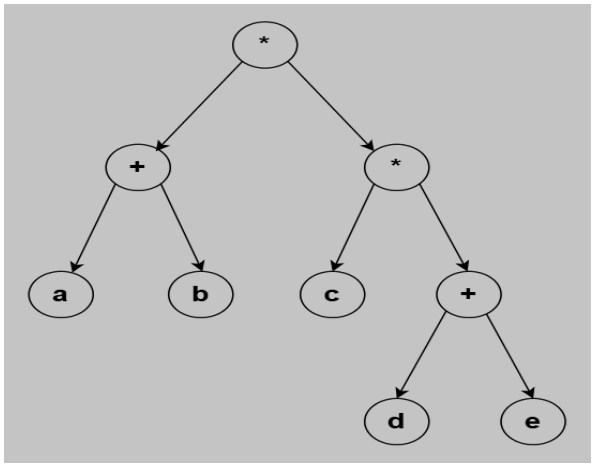
- ✓ For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).
- ✓ Time Complexity: O(1), In the push function a single element is inserted at the last position.
- ✓ Auxiliary Space: O(1), As no extra space is being used.

Some Applications of Stacks:

- ✓ The Stack is used as a Temporary Storage Structure for recursive operations.
- ✓ Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.
- ✓ We can manage function calls using Stacks.
- ✓ Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.
- ✓ Stacks are also helpful in converting infix expressions to postfix expressions.
- ✓ Stacks allow us to check the expression's syntax in the programming environment.
- ✓ We can match parenthesis using Stacks.
- ✓ Stacks can be used to reverse a String.
- ✓ Stacks are helpful in solving problems based on backtracking.
- ✓ We can use Stacks in depth-first search in graph and tree traversal.
- ✓ Stacks are also used in Operating System functions.
- ✓ Stacks are also used in UNDO and REDO functions in an edit.

Example:

Construction of a binary expression tree for infix notation (a+b)*(c*(d+e)).



Binary Tree

Infix Expression: ((a+b)*(c*(d+e)))
Prefix Expression: *+ab *c + de
Postfix Expression: ab+cde+**

1.3. QUEUE

Queue:

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
- A **Queue is a linear data structure**
- Queue is referred to be as **First In First Out list (LIFO).**
- For example, people waiting in line for a rail ticket form a queue.

Applications of Queue:

- Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering ofactions.
- There are various applications of queues:
  1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
  2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same ratebetween two processes) for eg. pipes, file IO, sockets.
  3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
  4. Queues are used to maintain the play list in media players in order to add and remove the songs fromthe play-list.
  5. Queues are used in operating systems for handling interrupts.

Basic Operations on Stacks:

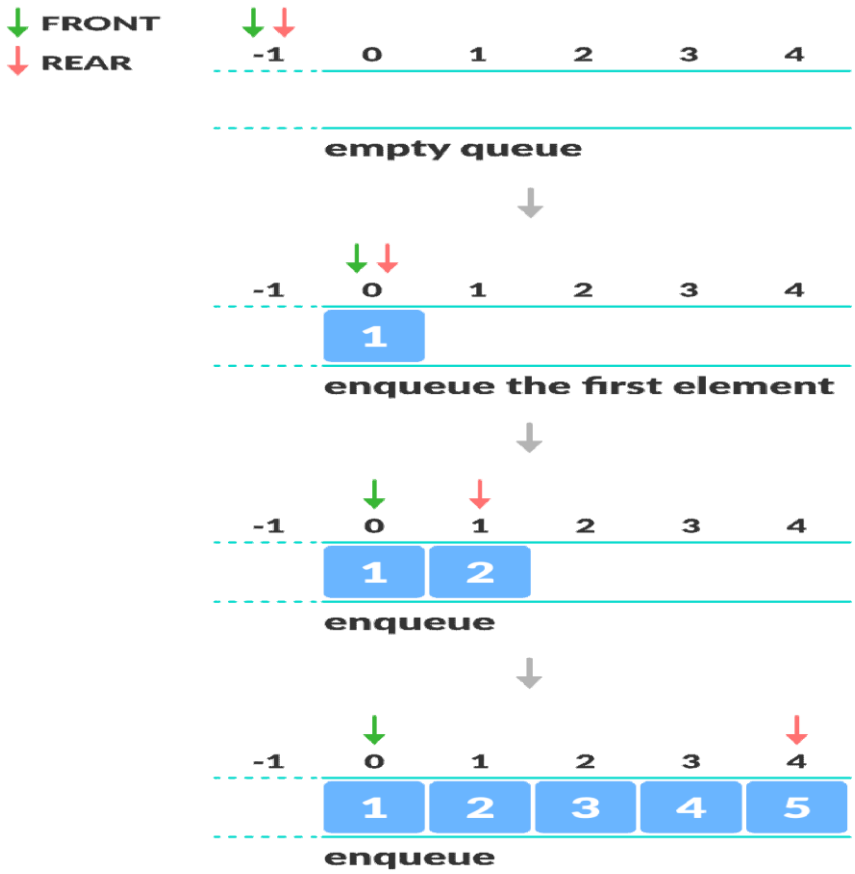
- The most fundamental operations in the stack include: **ENQUEUE ( ), DEQUEUE ( ).**
- These are allbuilt-in operations to carry out data manipulation and to check the status of the stack.

Insertion: ENQUEUE( )

- Enqueue() is an operation that inserts elements into the queue.
- The following is an algorithm that describesthe Enqueue () operation.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

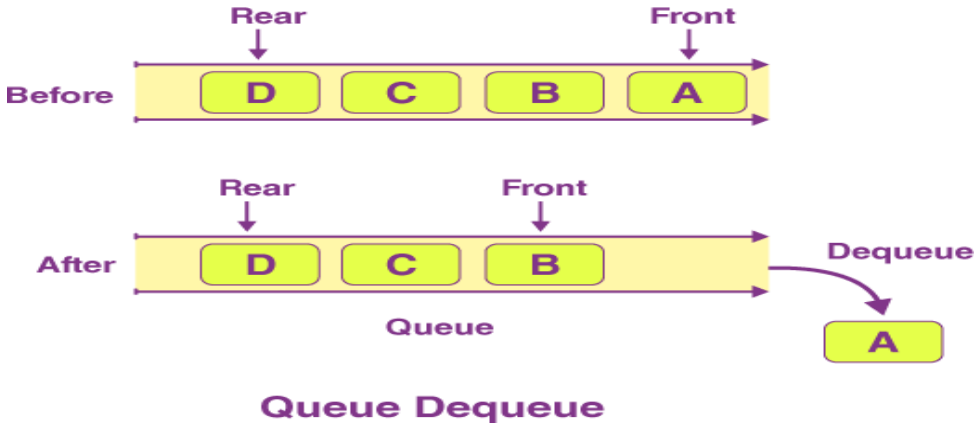
ENQUEUE( )



Deletion: DEQUEUE( )

- Dequeue() is an operation that delete elements from the queue.
- The following is an algorithm thatdescribes the dequeue () operation.

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```



Queue Dequeue

Characteristics of Queue:

- Queue can handle multiple data,
- We can access both ends,
- They are fast and flexible.

Features of Queue:

- Like stack, queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- peek( ) function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue:

- ✓ Task scheduling, resource allocation, message queues, print spooling, traffic management, customer service, and data buffering.
- ✓ Process scheduling, disk scheduling, memory management, IO buffer, pipes, call center phone systems, and interrupt handling.

Complexity Analysis of Queue Operations:

- Enqueue: O(1)
- Dequeue: O(1)
- Size: O(1)

Some Applications of Queues:

- ✓ Queues are generally used in the breadth search operation in Graphs.
- ✓ Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.
- ✓ Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.
- ✓ Priority Queues are utilized in file-downloading operations in a browser.
- ✓ Queues are also used to transfer data between peripheral devices and the CPU.
- ✓ Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

Advantages of Queue:

- ❖ A large amount of data can be managed efficiently with ease.
- ❖ Operations such as insertion and deletion can be performed and follow first in first out rule.
- ❖ Queues are useful when a particular service is used by multiple consumers.
- ❖ Queues are fast in speed for data inter-process communication.
- ❖ Queues can be used in the implementation of other data structures.

Disadvantages of Queue:

- ❖ The operations such as insertion and deletion of elements from the middle are time consuming.
- ❖ Limited Space.
- ❖ In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- ❖ Maximum size of a queue must be defined prior.

Types of Queue:

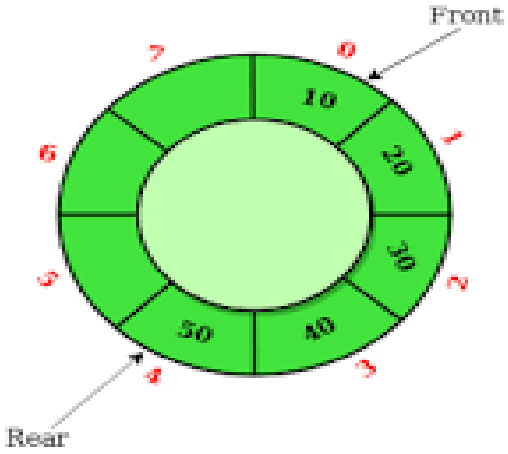
- There are four different types of queue that are listed as follows –
  - ✓ Simple Queue or Linear Queue
  - ✓ Circular Queue
  - ✓ Priority Queue
  - ✓ Double Ended Queue (or Deque)

Type 1: Simple Queue or Linear Queue:

- It is the most basic queue in which the insertion of an item is done at the front of the queue and deletion takes place at the end of the queue. Ordered collection of comparable data kinds. Queue structure is FIFO (First in, First Out).

Type 2: Circular Queue:

- A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.
- The operations are performed based on FIFO (First In First Out) principle.
- It is also called 'Ring Buffer'.



- A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

Operations on Circular Queue:

- Front: Get the front item from the queue.
- Rear: Get the last item from the queue.
- EnQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
- Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
- If it is full then display Queue is full.
- If the queue is not full then, insert an element at the end of the queue.
- deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
- Check whether the queue is Empty.
- If it is empty then display Queue is empty.
- If the queue is not empty, then get the last element and remove it from the queue.

Applications of a Circular Queue

- Memory management: circular queue is used in memory management.
- Process Scheduling: A CPU uses a queue to schedule processes.
- Traffic Systems: Queues are also used in traffic systems.
- page replacement algorithm,
- CPU Scheduling
- Inter-process communication

Abstract Data typesof Circular Queue:

- The following are the operations that can be performed on a circular queue.
    - enQueue()
    - deQueue()
    - front()
    - rear()

Advantages of Circular Queue:

- ✓ It provides a quick way to store FIFO data with a maximum size.
- ✓ Efficient utilization of the memory.
- ✓ Doesn't use dynamic memory.
- ✓ Simple implementation.
- ✓ All operations occur in O(1) constant time.

Disadvantages of Circular Queue:

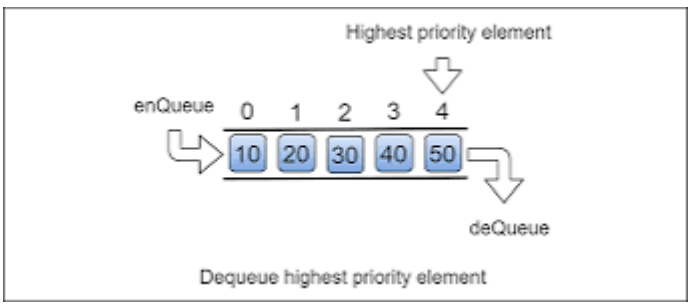
- ✓ In a circular queue, the number of elements you can store is only as much as the queue length, you have to know the maximum size beforehand.
- ✓ Some operations like deletion or insertion can be complex in circular queue.
- ✓ The implementation of some algorithms like priority queue can be difficult in circular queue.
- ✓ Circular queue has a fixed size, and when it is full, there is a risk of overflow if not managed properly.

Real-time Applications of Circular Queue:

- Months in a year: Jan –> Feb –> March –> and so on upto Dec–> Jan –> . . .
- Eating: Breakfast –> lunch –> snacks –> dinner –> breakfast –> and so on..
- Traffic Light is also a real-time application of circular queue.
- Clock is also a better example for the Circular Queue.

Type 3: Priority Queue:

- A priority queue is a type of queue that arranges elements based on their priority values.
- Elements with higher priority values are typically retrieved before elements with lower priority values.
- In a priority queue, each element has a priority value associated with it.



Properties of Priority Queue

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Operations of a Priority Queue:

- A typical priority queue supports the following operations:
- **Insertion in a Priority Queue**
    - When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right.
- **Deletion in a Priority Queue**
    - A max heap, the maximum element is the root node. And it will remove the element which has maximum priority first.

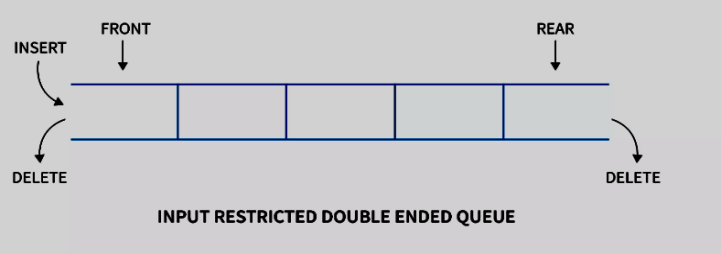
- **Peek in a Priority Queue**
  - This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

Application of Priority queue

- Dijkstra's Shortest Path Algorithm using priority queue:
- Prim's algorithm
- Data Compression
- Heap Sort
- Operating System
- Robotics etc.,

Type 4: Deque (or, Double Ended Queue):

- Insertion and removal of elements can either be performed from the front or the rear.
- It does not follow FIFO rule (First In First Out).
- Deque is a double-ended queue that is the implementation of the simple Queue.
- Still, the insertion and deletion of elements take place from both the ends.
- A deque in data structure is a linear data structure that does not follow the FIFO rule (First in first out) , that is, in a deque data structure, the data can be inserted and deleted from both front and rear ends.
- The representation of a deque in data structure is represented below -

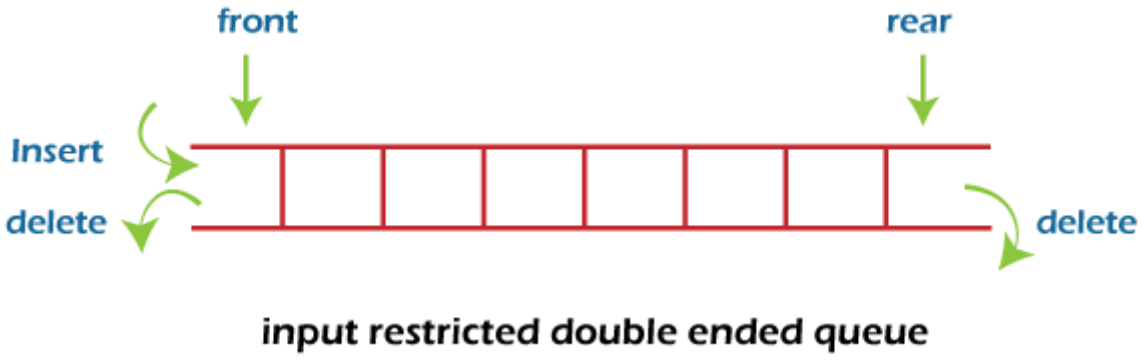


Types :

- ***Input restricted queue***
- ***Output restricted queue***

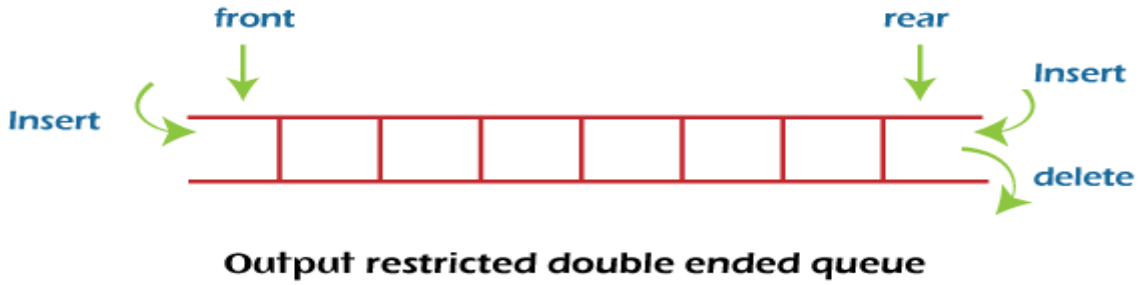
Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Output restricted Queue

- In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Advantages of Deque:

- ✓ Add and remove items from the both front and back of the queue.
- ✓ Deques are faster in adding and removing the elements to the end or beginning.
- ✓ The clockwise and anti-clockwise **rotation** operations are faster in a deque.
- ✓ Dynamic Size: Deques can grow or shrink dynamically.

Disadvantages of Deque:

- ✓ Priority Queues are that the enqueue and dequeue operations are slow
- ✓ A time complexity of O(log n).

Applications of Deque:

- ❖ Applied as both stack and queue, as it supports both operations.
- ❖ Storing a web browser's history.
- ❖ Storing a software application's list of undo operations.
- ❖ Job scheduling algorithm.

1.4. TREE

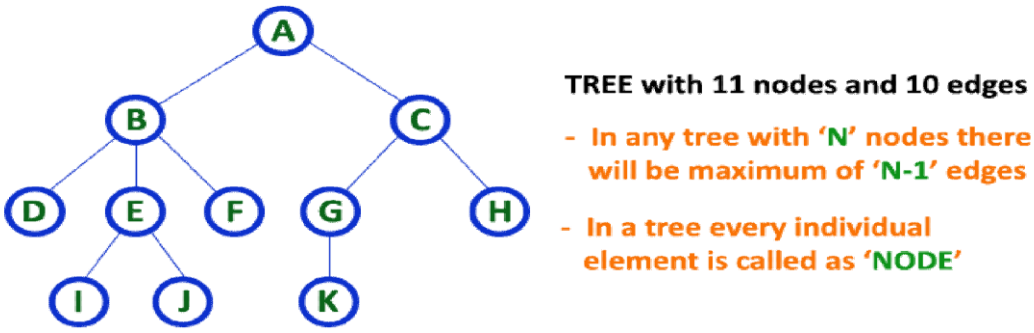
Tree:

- In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order.
- **A tree is a very popular non-linear data structure used in a wide range of applications.**
- A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structurerecursively

- In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.
- In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example:

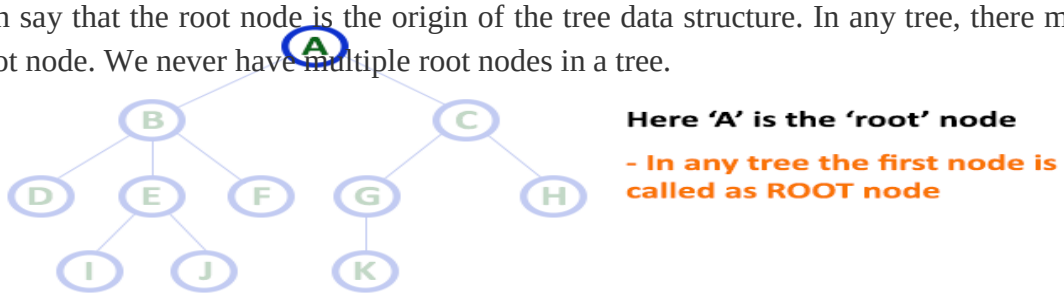


Tree Terminology:

- In a tree data structure, we use the following terminology...

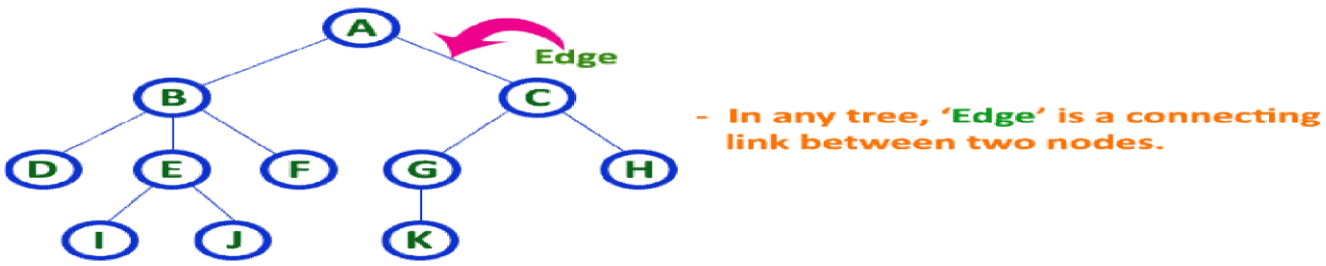
1. Root:

- In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



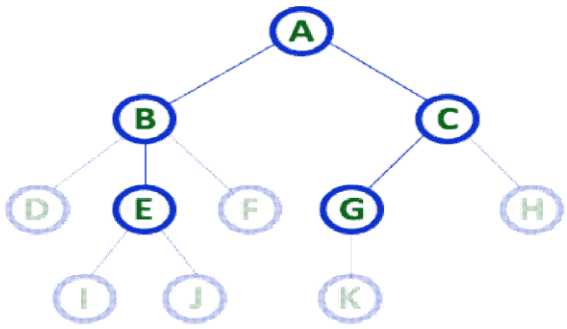
2. Edge:

- In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**'number of nodes there will be a maximum of '**N-1**' number of edges.



3. Parent:

- In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simplewords, the node which has a branch from it to any other node is called a parent node. Parent node can also bedefined as "**The node which has child / children**".

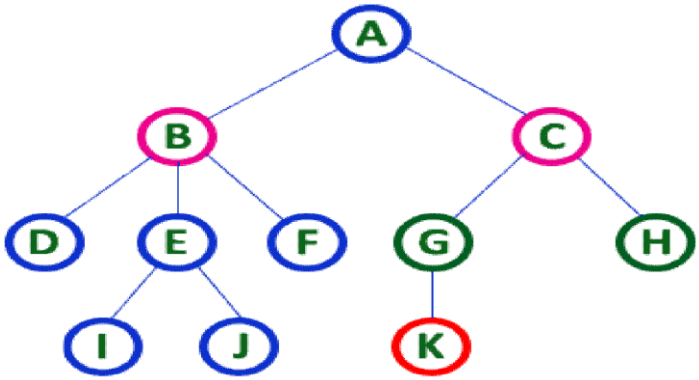


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child:

- In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node canhave any number of child nodes. In a tree, all the nodes except root are child nodes.

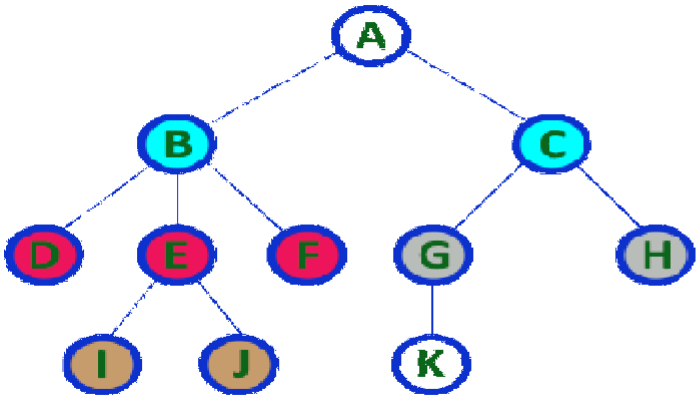


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings:

- In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple

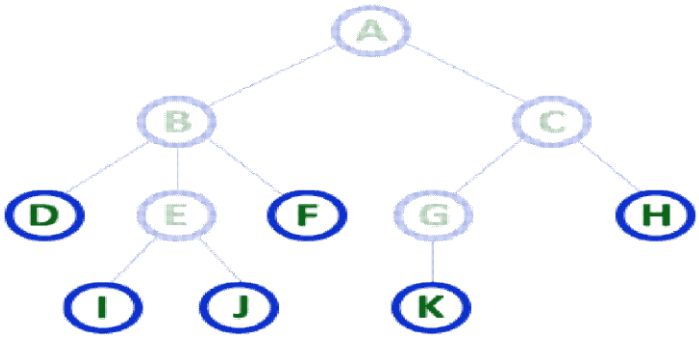


Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf:

- In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words,a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.

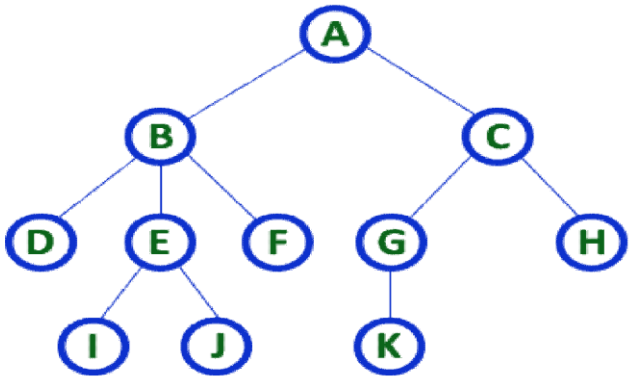


Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Degree:

- In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node amongall the nodes in a tree is called as '**Degree of Tree**'



Here Degree of B is 3

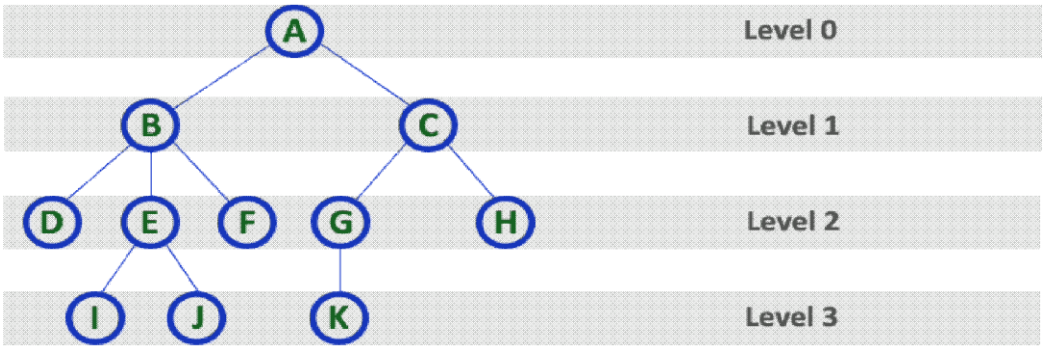
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

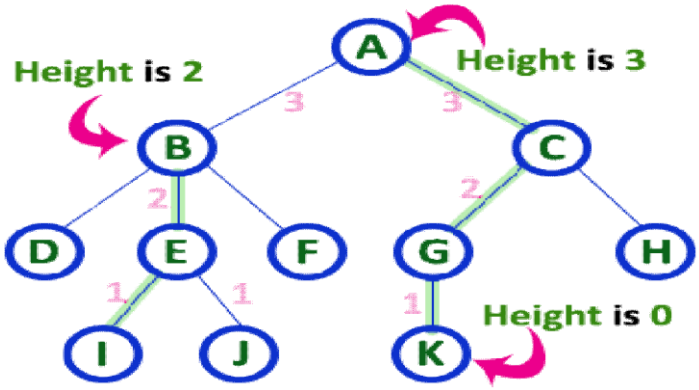
8. Level:

- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



9. Height:

- In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In atree, **height of all leaf nodes is '0'.**

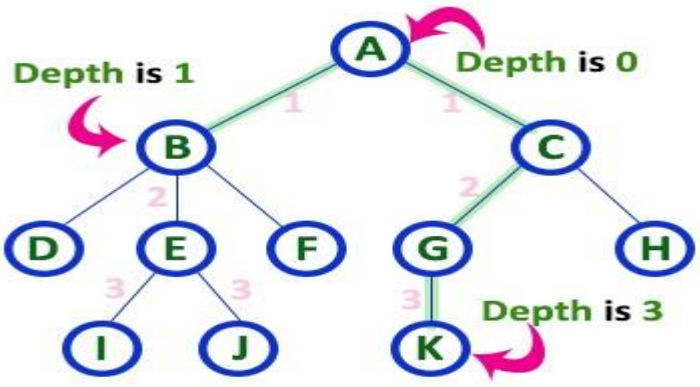


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

10. Depth:

- In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a

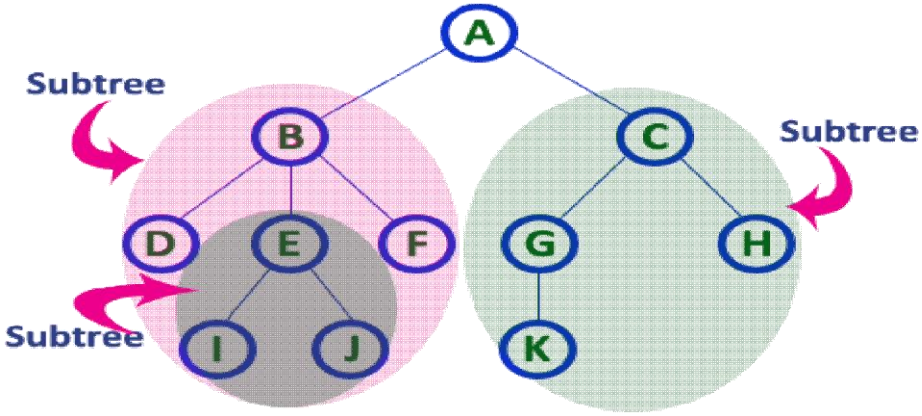


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

11. Sub Tree:

- In a tree data structure, each child from a node forms a subtree recursively. Every child node will subtree on its parent node.

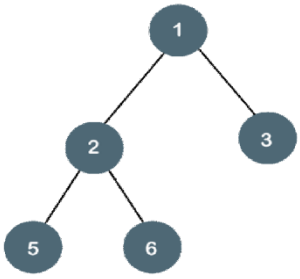


Types of Tree Data Structure:

- The following are the different types of trees data structures:
  - Binary Tree
  - Binary Search Tree (BST)
  - AVL Tree
  - B-Tree

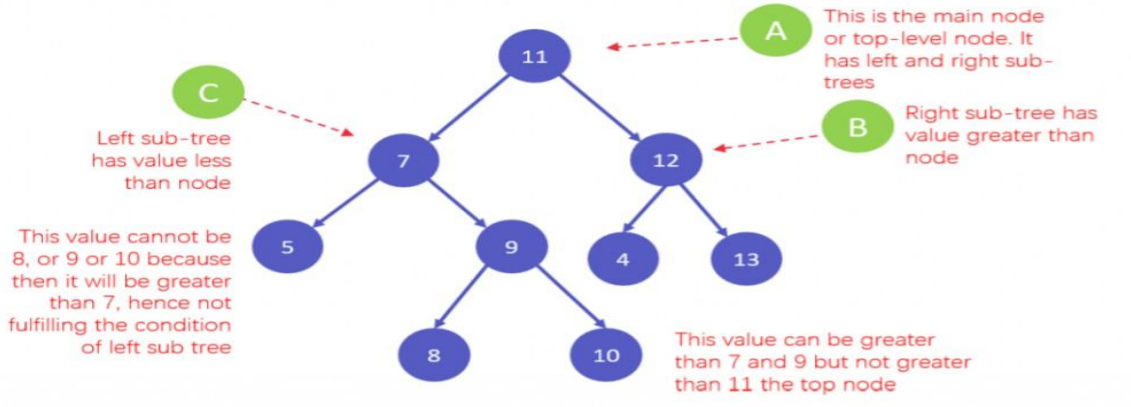
1. Binary Tree:

- A binary tree is a tree data structure in which each node can have 0, 1, or 2 children – left and right child.



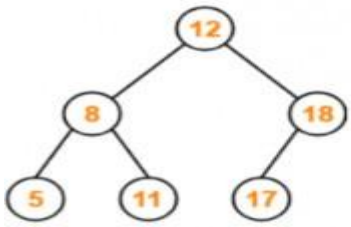
2. Binary Search Tree (BST):

- A binary search tree (BST) is also called an ordered or sorted binary tree in which the value at the left sub- tree is lesser than that of the root and the right subtree has a value greater than that of the root.
- Every binary search tree is a binary tree. However, not every binary tree is a binary search tree.
- What's the difference between a binary tree and a binary search tree? The most important difference between the two is that in a BST, the left child node's value must be less than the parent's while the right child node's value must be higher.

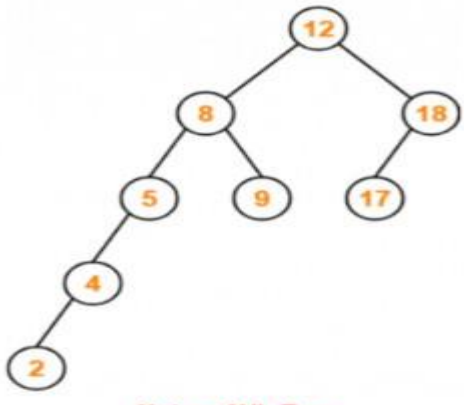


3. AVL Tree:

- AVL trees are a special kind of self-balancing binary search tree where the height of every node's left andright subtree differs by at most one.



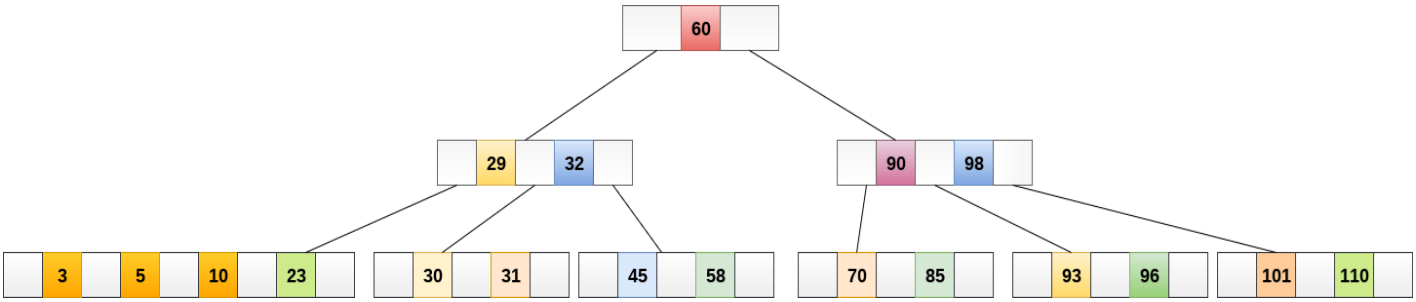
AVL Tree Example



Not an AVL Tree

4. B-Tree:

- B tree is a self-balancing search tree wherein each node can contain more than one key and more than two children. It is a special type of m-way tree and a generalized binary search tree. B-tree can store many keysin a single node and can have multiple child nodes. This reduces the height and enables faster disk access.



Basic Operation of Tree Data Structure:

- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal**:
    - **Preorder Traversal**
    - **In order Traversal**
    - **Post-order Traversal**

Tree Applications:

- ❖ Binary Search Trees (BSTs) are used to quickly check whether an element is present in a set or not.
- ❖ Heap is a kind of tree that is used for heap sort.
- ❖ A modified version of a tree called Tries is used in modern routers to store routing information.
- ❖ Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- ❖ Compilers use a syntax tree to validate the syntax of every program you write.

Need for Tree Data Structure:

1. To store information that naturally forms a hierarchy. For example, the file system on a computer:
2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Application of Tree Data Structure:

- ✓ File System
- ✓ Data Compression
- ✓ Compiler Design
- ✓ Database Indexing

Advantages of Tree Data Structure:

- ✓ Tree offer Efficient Searching Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- ✓ Trees provide a hierarchical representation of data, making it easy to organize and navigate large amounts of information.
- ✓ The recursive nature of trees makes them easy to traverse and manipulate using recursive algorithms.

Disadvantages of Tree Data Structure:

- ✓ Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to inefficient search times.
- ✓ Trees demand more memory space requirements than some other data structures like arrays and linked lists, especially if the tree is very large.
- ✓ The implementation and manipulation of trees can be complex and require a good understanding of the algorithms.

Some Applications of Trees:

- ❖ Trees implement hierarchical structures in computer systems like directories and file systems.
- ❖ Trees are also used to implement the navigation structure of a website.
- ❖ We can generate code like Huffman's code using Trees.
- ❖ Trees are also helpful in decision-making in Gaming applications.
- ❖ Trees are responsible for implementing priority queues for priority-based OS scheduling functions.
- ❖ Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.
- ❖ We can use Trees to store data keys for indexing for Database Management System (DBMS).
- ❖ Spanning Trees allows us to route decisions in Computer and Communications Networks.
- ❖ Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications

1.5. PRIORITY QUEUE

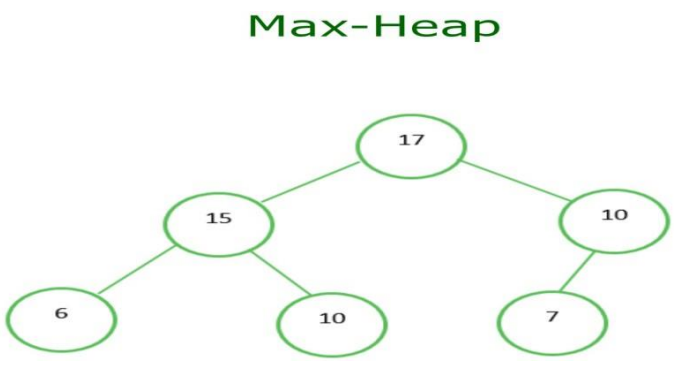
Priority Queue:

- ➢ Any data structure that supports the operations of search min (or max), insert, and delete min (or max, respectively) is called a priority queue. Priority queue concepts implemented using Heap and Heap Sorts.

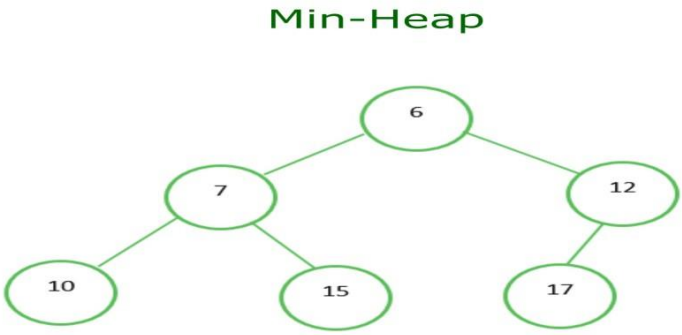
HEAP:

- ➢ Heap is a complete binary tree structure where each element satisfies a heap property.
- ➢ In a complete binary tree, all levels are full except the last level, i.e., nodes in all levels except the last level will have two children.
- ➢ The last level will be filled from the left. Here, each heap node stores a value key, which defines the relative position of that node inside the heap.
- ➢ Heap can be Max heap or Min heap.

Max Heap: Complete Binary tree having largest elements is at the root of the heap.



Min Heap: Complete Binary tree having smallest elements is at the root of the heap.



Algorithm for inserting elements into the Heap

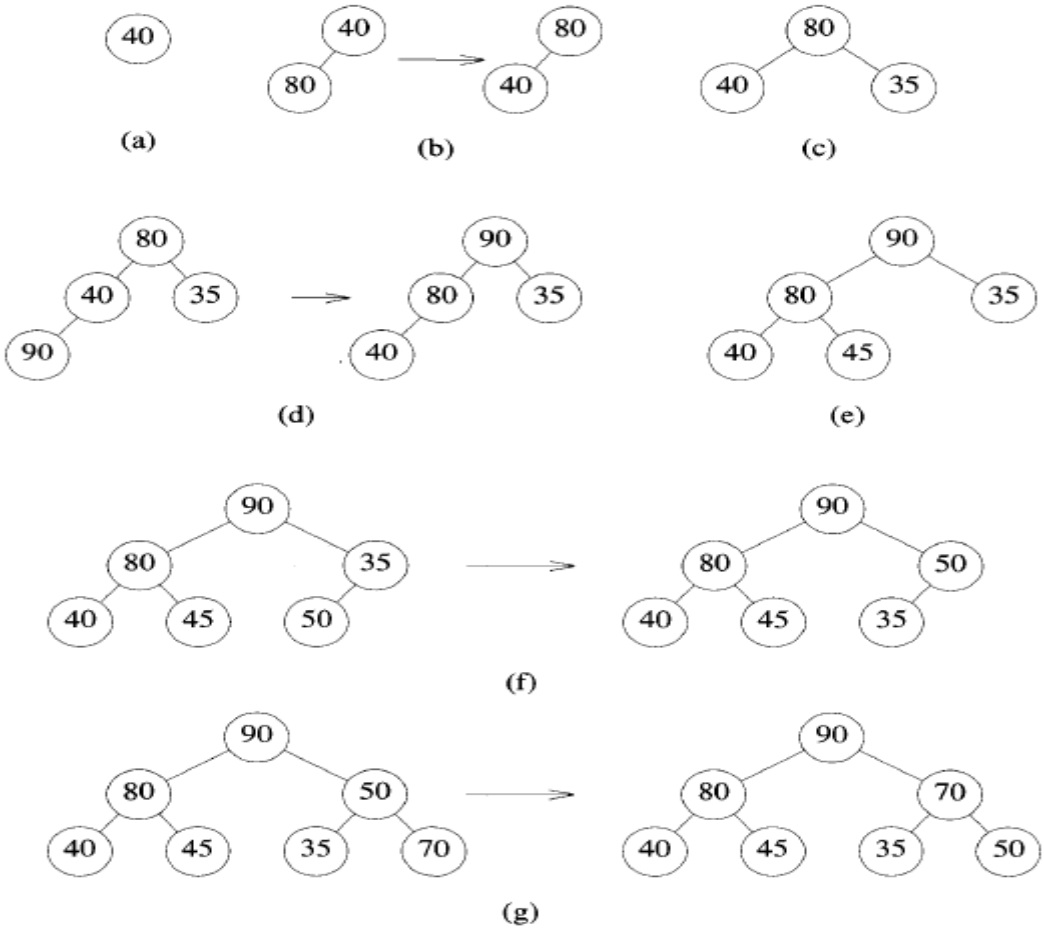
```
1   Algorithm Insert(a, n)
2   {
3       // Inserts a[n] into the heap which is stored in a[1 : n − 1].
4       i := n; item := a[n];
5       while ((i > 1) and (a[⌊i/2⌋] < item)) do
6       {
7           a[i] := a[⌊i/2⌋]; i := ⌊i/2⌋;
8       }
9       a[i] := item; return true;
10  }
```

Max Heap Construction Algorithm:

- An algorithm for max heap by inserting one element at a time.
- At any point of time, heap must maintainits property.
- While insertion, we also assume that we are inserting a node in an already heapified tree.

```
Step 1 - Create a new node at the end of heap.
Step 2 - Assign new value to the node.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.
```

Forming a Heap from the set {40, 80, 35, 90, 45, 50, 70}




```
1   Algorithm Adjust(a, i, n)
2   // The complete binary trees with roots 2i and 2i + 1 are
3   // combined with node i to form a heap rooted at i. No
4   // node has an address greater than n or less than 1.
5   {
6       j := 2i; item := a[i];
7       while (j ≤ n) do
8       {
9           if ((j < n) and (a[j] < a[j + 1])) then j := j + 1;
10              // Compare left and right child
11              // and let j be the larger child.
12          if (item ≥ a[j]) then break;
13              // A position for item is found.
14          a[⌊j/2⌋] := a[j]; j := 2j;
15      }
16      a[⌊j/2⌋] := item;
17  }
```

```
1   Algorithm DelMax(a, n, x)
2   // Delete the maximum from the heap a[1 : n] and store it in x.
3   {
4       if (n = 0) then
5       {
6           write ("heap is empty"); return false;
7       }
8       x := a[1]; a[1] := a[n];
9       Adjust(a, 1, n − 1); return true;
10  }
```

Max Heap Deletion Algorithm:

- Algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to removethe Maximum (or minimum) value.

```
Step 1 - Remove root node.
Step 2 - Move the last element of last level to root.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.
```

HEAP SORT:

- Heap sort is a popular and efficient sorting algorithm.
- The concept of heap sort is to eliminate the elementsone by one from the heap part of the list, and then insert them into the sorted part of the list.

Algorithm

```
1.  HeapSort(arr)
2.  BuildMaxHeap(arr)
3.  for i = length(arr) to 2
4.      swap arr[1] with arr[i]
5.      heap_size[arr] = heap_size[arr] ? 1
6.      MaxHeapify(arr,1)
7.  End
```

Heapify

BuildMaxHeap(arr)

```
1.  BuildMaxHeap(arr)
2.    heap_size(arr) = length(arr)
3.    for i = length(arr)/2 to 1
4.  MaxHeapify(arr,i)
5.  End
```

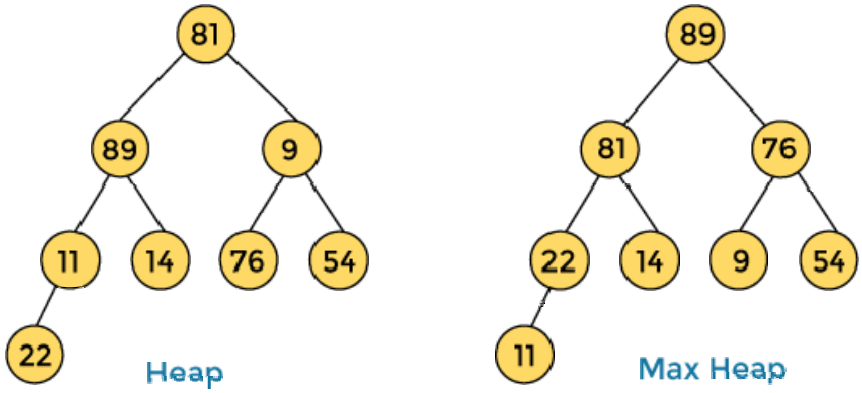
MaxHeapify(arr,i)

```
1.      MaxHeapify(arr,i)
2.      L = left(i)
3.      R = right(i)
4.      if L ? heap_size[arr] and arr[L] > arr[i]
5.      largest = L
6.      else
7.      largest = i
8.      if R ? heap_size[arr] and arr[R] > arr[largest]
9.      largest = R
10.     if largest != i
11.     swap arr[i] with arr[largest]
12.     MaxHeapify(arr,largest)
13.     End
```

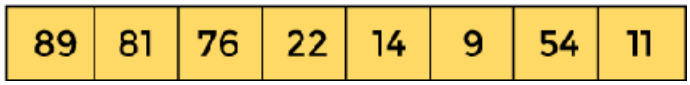
Working of Heap sort Algorithm:

- In heap sort, basically, there are two phases involved in the sorting of elements.
- By using the heap sortalgorithm, they are as follows -
  - The first step includes the creation of a heap by adjusting the elements of the array.
  - After the creation of heap, now remove the root element of the heap repeatedly by shifting it to theend of the array, and then store the heap structure with the remaining elements.

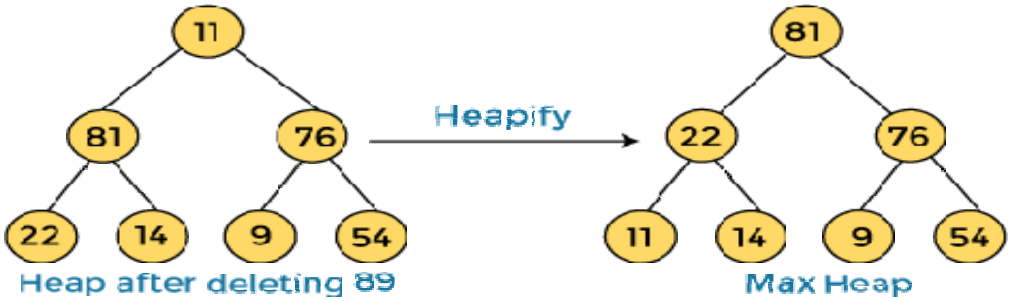
- Construct a heap from the given array and convert it into max heap.



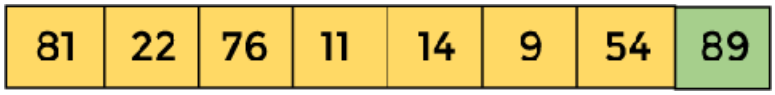
- After converting the given heap into max heap, the array elements are -



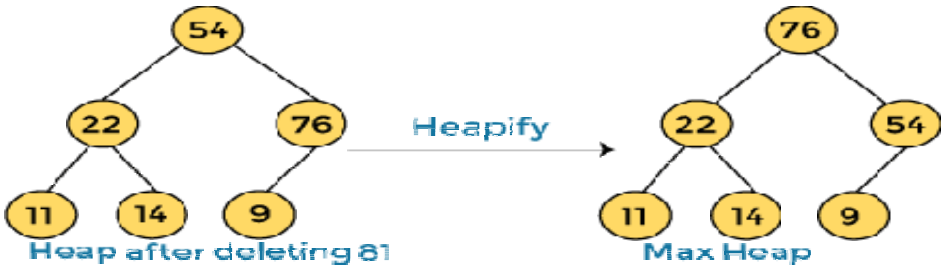
- Next, we have to delete the root element **(89)** from the max heap.
- To delete this node, we have to swap itwith the last node, i.e. **(11).**
- After deleting the root element, we again have to heapify it to convert it intomax heap.



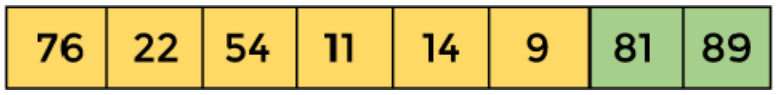
- After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



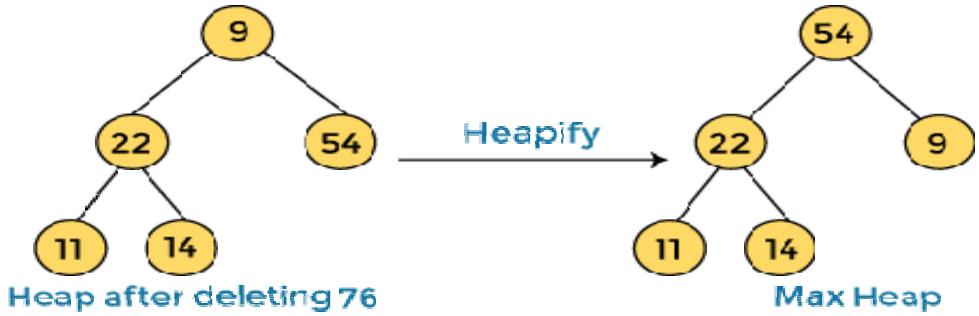
- In the next step, again, we have to delete the root element **(81)** from the max heap.
- To delete this node, we have to swap it with the last node, i.e. **(54).**
- After deleting the root element, we again have to heapify it to convert it into max heap.



- After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

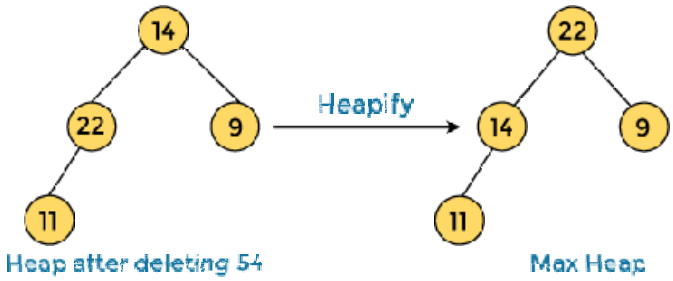


- In the next step, we have to delete the root element **(76)** from the max heap again.
- To delete this node, we have to swap it with the last node, i.e. **(9).**
- After deleting the root element, we again have to heapify it to convert it into max heap.

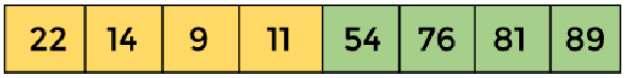


- After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are

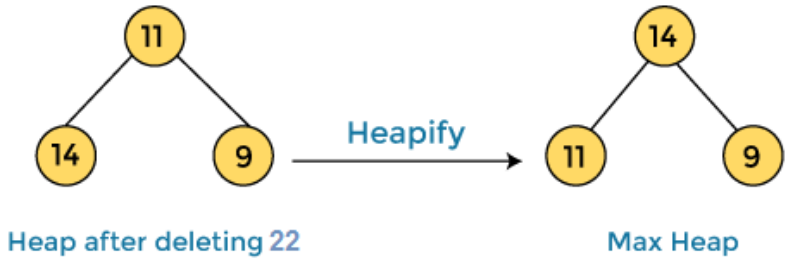
- In the next step, again we have to delete the root element **(54)** from the max heap.
- To delete this node, we have to swap it with the last node, i.e. **(14).**
- After deleting the root element, we again have to heapify it to convert it into max heap.



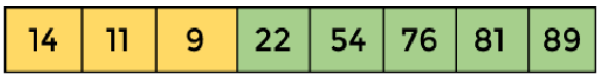
- After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



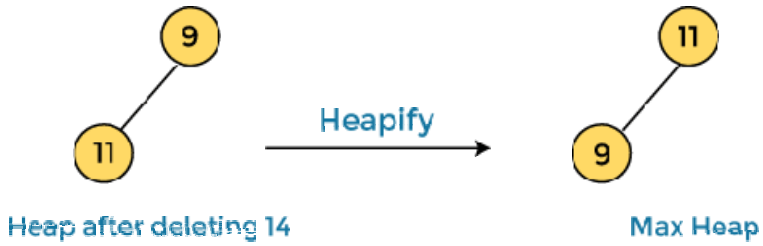
- In the next step, again we have to delete the root element **(22)** from the max heap.
- To delete this node, we have to swap it with the last node, i.e. **(11).**
- After deleting the root element, we again have to heapify it to convert it into max heap.



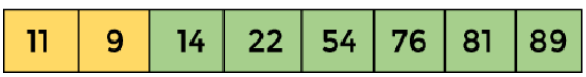
- After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



- In the next step, again we have to delete the root element **(14)** from the max heap.
- To delete this node, we have to swap it with the last node, i.e. **(9).**
- After deleting the root element, we again have to heapify it to convert it into max heap.

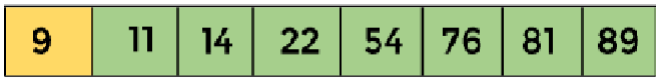


- After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are



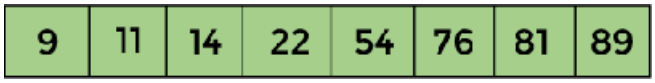
- In the next step, again we have to delete the root element **(11)** from the max heap.
- To delete this node, we have to swap it with thelast node, i.e. **(9).**
- After deleting the root element, we again have to heapify it to convert it into max heap.

- After swapping the array element **11** with **9,** the elements of array are -



- Now, heap has only one element left. After deleting it, heap will be empty.

- After completion of sorting, the array elements are -



- Now, the array is completely sorted.

Heap sort complexity:

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

1. Time Complexity:

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**

- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity:

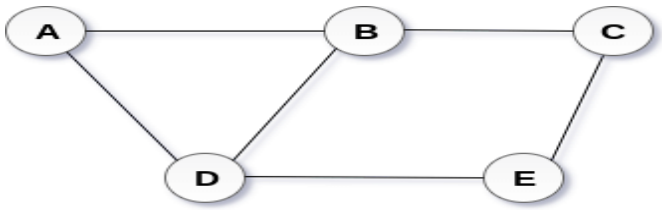
Space Complexity	O(1)
Stable	N0

1.6. GRAPH

- A graph can be defined as group of vertices and edges that are used to connect these vertices.
- A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them insteadof having parent child relationship.

Definition:

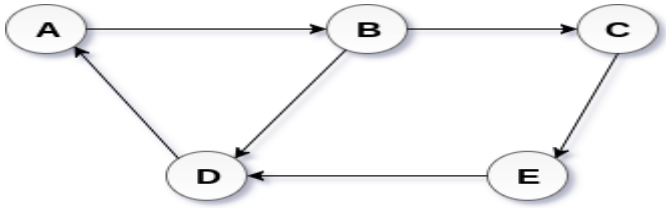
- A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G)represents the set of edges which are used to connect these vertices.
- A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

Directed and Undirected Graph:

- A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them.
- An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.



Directed Graph

Graph Terminology:

Path:

- A path can be defined as the sequence of nodes that are followed in order to reach some terminal node Vfrom the initial node U.

Closed Path:

- A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if$V_0=V_N$.

Simple Path:

- If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simplepath.

Cycle:

- A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph:

- A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are noisolated nodes in connected graph.

Complete Graph:

- A complete graph is the one in which every node is connected with all other nodes. A complete graphcontain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph:

- In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge ecan be given as w (e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph:

- A digraph is a directed graph in which each edge of the graph is associated with some direction and thetraversing can be done only in the specified direction.

Loop:

- An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes:

- If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacentnodes.

Degree of the Node:

- A degree of a node is the number of edges that are connected with that node. A node with degree 0 is calledas isolated node.

Basic Operations on Graphs:

- Below are the basic operations on the graph:
  - ✓ Insertion of Nodes/Edges in the graph – Insert a node into the graph.
  - ✓ Deletion of Nodes/Edges in the graph – Delete a node from the graph.
  - ✓ Searching on Graphs – Search an entity in the graph.
  - ✓ Traversal of Graphs – Traversing all the nodes in the graph.

Graph representation:

- There are two ways to store Graphs into the computer's memory:
  - **Sequential representation** (or, Adjacency matrix representation)
  - **Linked list representation** (or, Adjacency list representation)
- In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked listrepresentation, there is a use of an adjacency list to store the graph.

Sequential representation:

- In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

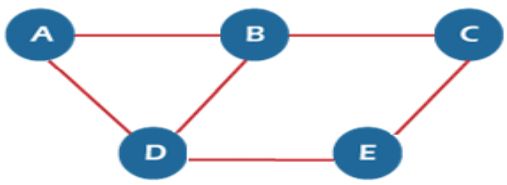
- If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

- An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

  a_{ij} = 1 {if there is a path exists from V_i to V_j}

  a_{ij} = 0 {Otherwise}

- It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes,whereas 1 represents the existence of a path between two edges.
- If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be0.
- Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

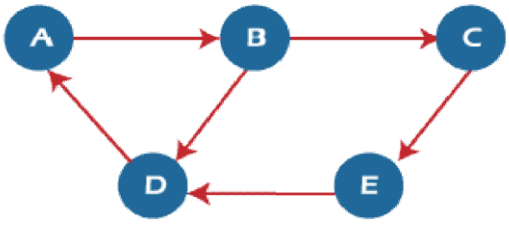
- In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

- There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

Adjacency matrix for a directed graph:

- In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

- Consider the below-directed graph and try to construct the adjacency matrix of it.

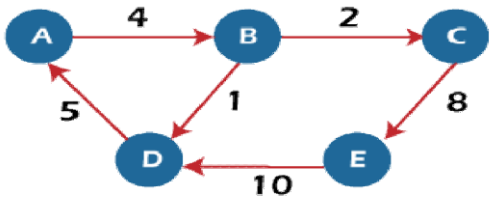


Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

- It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge.
- The weights on the graph edges will be represented as the entries of the adjacency matrix.
- We can understand it with the help of an example.
- Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



weighted Directed Graph

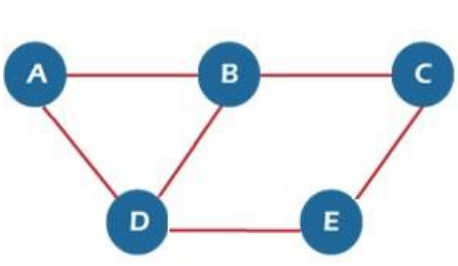
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

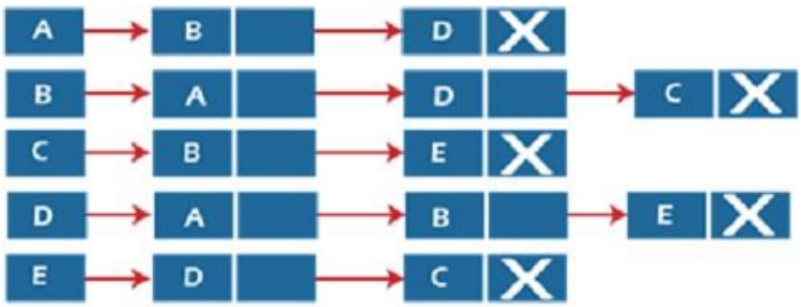
- In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

Linked list representation:

- An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.
- Let's see the adjacency list representation of an undirected graph.

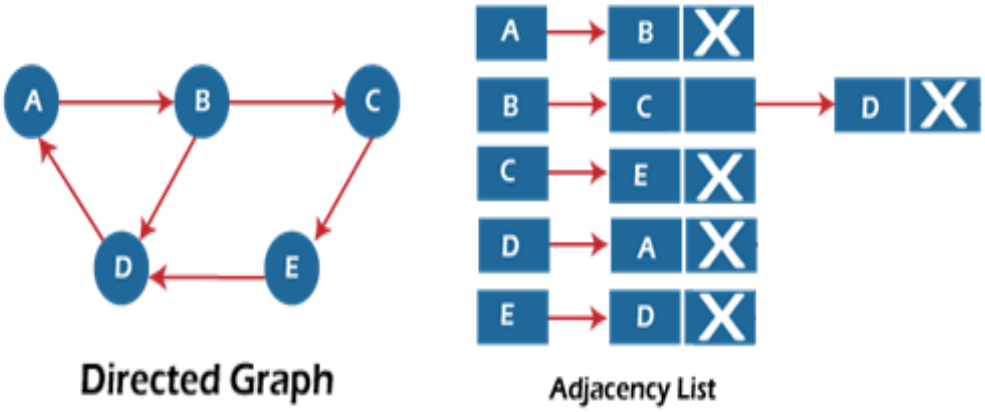


Undirected Graph

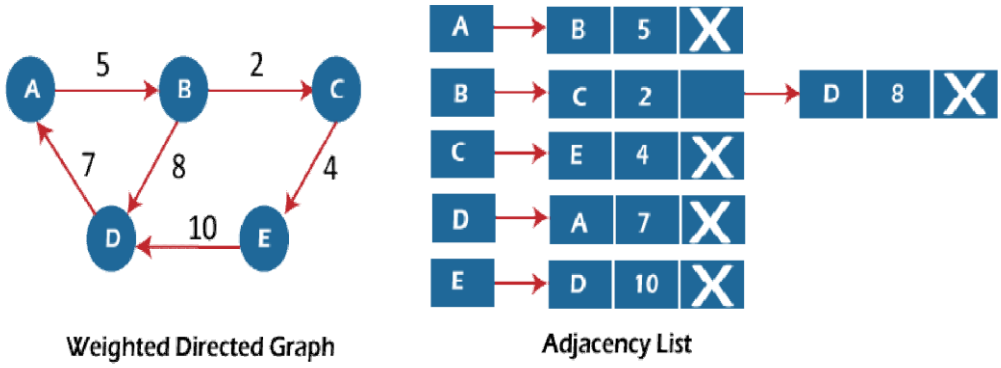


Adjacency List

- In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D.
- These nodes are linked to nodes A in the given adjacency list.
- An adjacency list is maintained for each node present in the graph, which stores the node value and a pointerto the next adjacent node to the respective node.
- If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.
- The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.
- Now, consider the directed graph, and let's see the adjacency list representation of that graph.



- For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.
- Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



- In the case of a weighted directed graph, each node contains an extra field that is called the weight of thenode.
- In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Some Applications of Graphs:

- ✓ Graphs help us represent routes and networks in transportation, travel, and communication applications.
- ✓ Graphs are used to display routes in GPS.
- ✓ Graphs also help us represent the interconnections in social networks and other network-based applications.
- ✓ Graphs are utilized in mapping applications.
- ✓ Graphs are responsible for the representation of user preference in e-commerce applications.

- ✓ Graphs are also used in Utility networks in order to identify the problems posed to local or municipal corporations.
- ✓ Graphs also help to manage the utilization and availability of resources in an organization.
- ✓ Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.
- ✓ Graphs are also used in robotic motions and neural networks.

1.7. WHAT IS AN ALGORITHM?

Algorithm:

- An algorithm is a procedure used for solving a problem or performing a computation.
- Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software- based routines.
- Algorithms are widely used throughout all areas of IT.
- In mathematics and computer science, an algorithm usually refers to a small procedure that solves a recurrent problem.
- Algorithms are also used as specifications for performing data processing and play a major role in automated systems.
- An algorithm could be used for sorting sets of numbers.

Definition: An algorithm is a finite set of instructions that is followed, accomplishes a particular task. In addition all algorithms must satisfy the following criteria:

1. **Input:** Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced.
3. **Definiteness:** Each instruction is clear and unambiguous.
4. **Finiteness:** If we trace out the instructions of an algorithm, then for all cases the algorithm terminates after a finite number of steps.
5. **Effectiveness:** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is also must be feasible.

- Algorithms that are definite and effective are also called computational procedures.
- One important example of computational procedures is the operating system of a digital computer.
- This procedure is designed to control the execution of jobs, in such a way that when no jobs are available, it does not terminate but continues in waiting state until a new job is entered.
- The study of algorithms includes many important and active areas of research.

Issues or Study of Algorithm (Issues in witting algorithm):

- Due to complicities of the Issues involved, a branch of computer engineering known as Software Engineering has emerged, where the main concern is software quality control.
- There are five distinct areas of study one can identify:

1. To devise algorithms: Creating an algorithm is an art which may never be fully automated. Important design techniques such as linear, nonlinear, useful in fields other than computer science such as operations research and electrical engineering.

2. To express an algorithm: An algorithm can be expressed in various ways: flow-chart, pseudo-code, program and like. Out of these only the program in certain programming language are acceptable to a computer. There are different ways by which an algorithm can be expressed p-casual or native programming structured programming, object-oriented programming, use of recursions and so n.

3. To validate algorithms: Once an algorithm is devised it is necessary to show that it computes the correct answer for all possible legal inputs. This process is called algorithm validation. The purpose of validation is to assure us that this algorithm will work correctly independently of the issues concerning

the programming language it will eventually be written in.

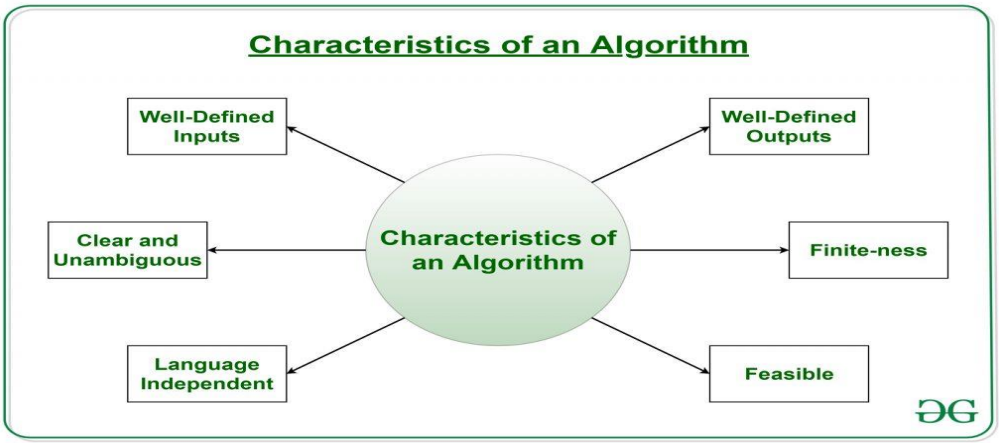
4. To analyze algorithms: This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory to hold the program and data. Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

5. To test a program: Testing a program consists of two-parts-debugging and profiling. It is importen to be familiar with the debugging of a program, though what we refer here is a more systematic approach to debugging. An interesting observation about debugging made by E.W.Dijkstra, a pioneer computer scientist from Netherlands, is worth quoting here:"debugging can only point to the presence of errors and newer their absence."

Need for algorithms:

- Algorithms are necessary for solving complex problems efficiently and effectively.
- They help to automate processes and make them more reliable, faster, and easier to perform.
- Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
- They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

Characteristics of an Algorithm:



- As one would not follow any written instructions to cook the recipe, but only the standard one.
- Similarly, not all written instructions for programming are an algorithm.
- For some instructions to be an algorithm, it must have the following characteristics:
  - **Clear and Unambiguous**: The algorithm should be 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. It may or may not take input.
  - **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
  - **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
  - **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with 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 the same, as expected.

- **Input**: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.
- **Output**: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- **Finiteness:** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In an 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.
- Understanding complex logic through algorithms can be very difficult.
- Branching and looping statements are difficult to show in Algorithms **(imp)**.

1.8. ALGORITHM SPECIFICATION

- Algorithm can be described in three ways:
  - ✓ Natural language: implement a natural language like English. When this way is choose care should be taken, we ensure that each & every statement is definite.
  - ✓ Flow charts: Graphic representations denoted flowcharts, only if the algorithm is small and simple.
  - ✓ Pseudo code Method: In this method, we should typically describe algorithm as program, which resembles language like Pascal & ALGOL.
- Pseudo code is a kind of structured English for described algorithm, it allows the designer to focus on the logic of the algorithm without being distracted (diverted) by details of language syntax.
- At the same time, the pseudo code needs to be complete.
- It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line –into source code.

Pseudocode Conventions:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces : { and }. A compound statement can be represented as a block.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context.

Ex: node = record
```
{
   datatype_1 data_1;
        .
        .
   datatype_n data_n;
   node *link;
}
```
4. Assignment of values to variable is done using the assignment statement <variable>:=<expression>;
5. There are two Boolean values true and false. In order to produce these values, the logical operators and or and not and the relational operators <,≤,=, ≠, ≥, and > are provided.
6. Elements of multidimensional arrays are accessed using [and]. Ex. A[i,j]
7. The following looping statements are employed : for, while and repeat until. The while loop takes the following form:

```
while <condition> do
{
        <statement 1>
        .
        .
        <statement n>
}
```

8. A conditional statement has the following forms:
   if <condition> then <statement>
   if <condition> then <statement 1> else <statement 2>
   Here <condition> is a Boolean expression and <statement>, <statement1> and <statement 2> are arbitrary statements (simple or compound).
9. Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.
10. There is only one type of procedure: Algorithm. An algorithm consists of a heading and a body. The heading takes the form
    Algorithm Name (<parameter list>)

Where Name is the name of the procedure and (<parameter list>) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm my or may not return any values.

Ex: Algorithm finds and returns the maximum of n given numbers:

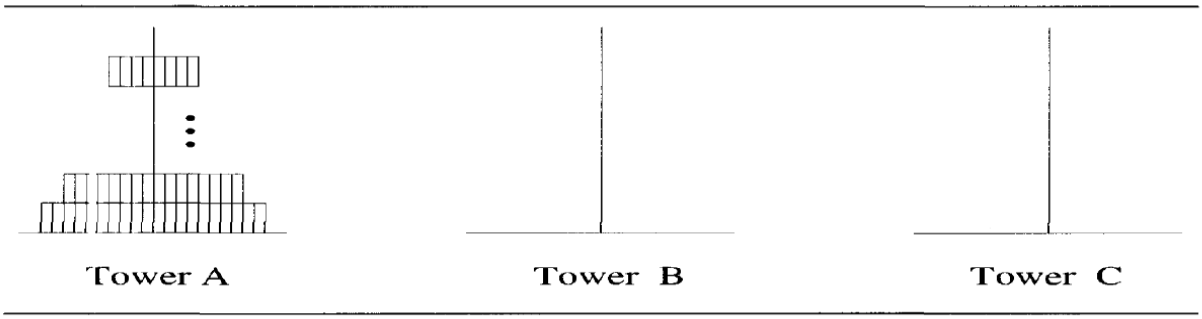
```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

Recursive Algorithms:

- A recursive function is a function that is defined in terms of itself. An algorithm that calls itself isdirect recursive.
- Algorithm A is said to indirect recursive if it calls another algorithm which in turn calls

A. These recursive mechanisms are extremely powerful, but even more important.

Example [Towers of Hanoi] The Towers of Hanoi puzzle is fashioned after the ancient Tower of Brahma ritual (see Figure 1.1). According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top. Besides this tower there were two other diamond towers (labeled B and C). Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower C for intermediate storage. As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk. According to legend, the world will come to an end when the priests have completed their task.



Towers of Hanoi

A very elegant solution results from the use of recursion. Assume that the number of disks is n. To get the largest disk to the bottom of tower B, we move the remaining $n-1$ disks to tower C and then move the largest to tower B. Now we are left with the task of moving the disks from tower C to tower B. To do this, we have towers A and B available. The fact that tower B has a disk on it can be ignored as the disk is larger than the disks being moved from tower C and so any disk can be placed on top of it. The recursive nature of the solution is apparent from Algorithm 1.3. This algorithm is invoked by TowersOfHanoi(n,A,B,C). Observe that our solution for an n-disk problem is formulated in terms of solutions to two $(n-1)$-disk problems.

Towers of Hanoi

```
1   Algorithm TowersOfHanoi(n, x, y, z)
2   // Move the top n disks from tower x to tower y.
3   {
4       if (n ≥ 1) then
5       {
6           TowersOfHanoi(n − 1, x, z, y);
7           write ("move top disk from tower", x,
8           "to top of tower", y);
9           TowersOfHanoi(n − 1, z, y, x);
10      }
11  }
```

1.9. PERFORMANCE ANALYSIS

Performance Analysis:

- Performance of an algorithm means predicting the resources which are required to an algorithm toperform its task.
- That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithmto solve that problem.
- We compare algorithms with each other which are solving the same problem, to select the best algorithm.
- To compare algorithms, we use a set of parameters or set of elements like memory required by thatalgorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

- Generally, the performance of an algorithm depends on the following elements...
  - ✓ Whether that algorithm is providing the exact solution for the problem?
  - ✓ Whether it is easy to understand?
  - ✓ Whether it is easy to implement?
  - ✓ How much space (memory) it requires to solve the problem?
  - ✓ How much time it takes to solve the problem? Etc.,

- When we want to analyze an algorithm, we consider only the space and time required by that particularalgorithm and we ignore all the remaining elements.
- Based on this information, performance analysis of an algorithm can also be defined as follows...

- Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.
- Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space
2. Time required to complete the task of that algorithm (Time Complexity).

1.9.1. Space Complexity:

- When we design an algorithm to solve a problem, it needs some computer memory to complete itsexecution. For any algorithm, memory is required for the following purposes...
  - To store program instructions.
  - To store constant values.
  - To store variable values.
  - And for few other things like function calls, jumping statements etc,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

- Generally, when a program is under execution it uses the computer memory for THREE reasons.
- They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

Note - When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.
That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

Example 1

```
int square(int a)
{
        return a*a;
}
```

- In the above piece of code, it requires 2 bytes of memory to store variable **'a'** and another 2 bytes of memory is usedfor **return value**.
- **That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed forany input value of 'a'.**
- **This space complexity is said to be *Constant Space Complexity*.**

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code...

Example 2

```
int sum(int A[ ], int n)
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

In the above piece of code it requires
'n*2' bytes of memory to store array variable **'a[ ]'**
2 bytes of memory for integer parameter **'n'**
4 bytes of memory for local integer variables **'sum'** and **'i'** (2 bytes each)
2 bytes of memory for **return value**.

- That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount ofmemory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately.
- This type of space complexity is said to be *Linear Space Complexity*.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

1.9.2. Time Complexity

- Every algorithm requires some amount of computer time to execute its instruction to perform the task. Thiscomputer time required is called time complexity.
- The time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
5. **Input** data

Note - When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

- Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system.

To solve this problem, we must assume a model machine with a specific configuration.
So that, we can able to calculate generalized timecomplexity according to that model machine.

- To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

- Now, we calculate the time complexity of following example code by using the above-defined modelmachine...
- Consider the following piece of code...

Example 1

```
int sum(int a, int b)
{
  return a+b;
}
```

- In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution.
- And it does not change based on the input values of a and b.
- That means for all input values, it requires the same amount of time i.e. 2 units.

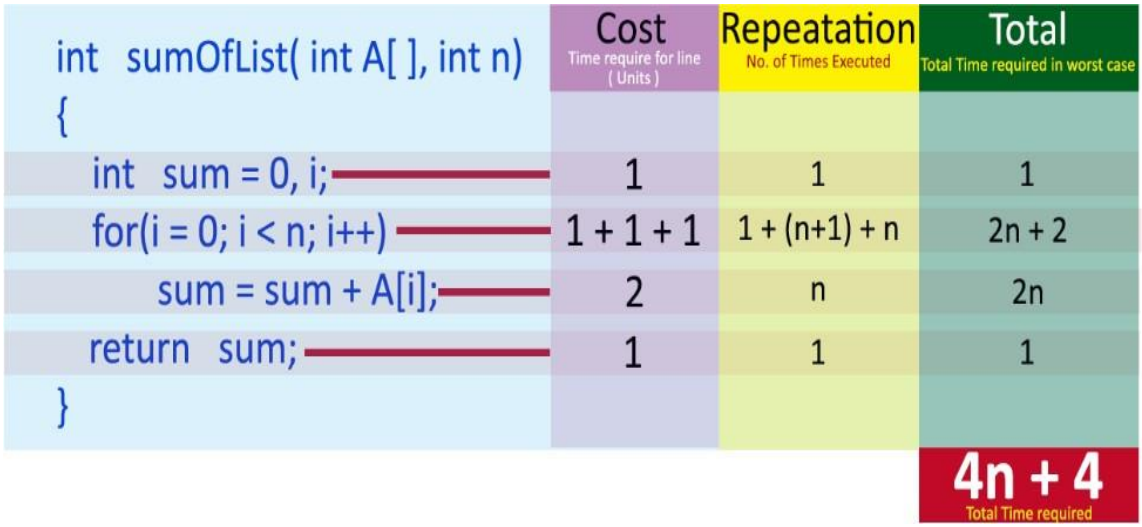
If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

Example 2

```
int sum(int A[], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
  return sum;
}
```

For the above code, time complexity can be calculated as follows...



- In above calculation
  - ✓ **Cost** is the amount of computer time required for a single operation in each line.
  - ✓ **Repeatation** is the amount of computer time required by each operation for all its repeatations.
  - ✓ **Total** is the amount of computer time required by each operation to execute.
- So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed.And it changes based on the **n** value.
- If we increase the **n** value then the time required also increases linearly.

- **Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.**

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

1.10. ASYMPTOTIC NOTATIONS

Asymptotic Notations:

- To perform analysis of an algorithm, we need to calculate the complexity of that algorithm.
- But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required.
- So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm.
- We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

- For example, consider the following time complexities of two algorithms...
  - ✓ **Algorithm 1: $5n^2 + 2n + 1$**
  - ✓ **Algorithm 2 : $10n^2 + 8n + 3$**

- Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data(i.e. **'n'** value). In above two time complexities, for larger value of **'n'** the term **'2n + 1'** in algorithm 1 hasleast significance than the term **'$5n^2$'**, and the term **'8n + 3'** in algorithm 2 has least significance than the term **'$10n^2$'**.
- Here, for larger value of **'n'** the value of most significant terms (5n2 and **$10n^2$)** is very larger than the value of least significant terms (2n + **1** and **8n + 3)**. So for larger value of **'n'** we ignore the least

significant termsto represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, THREE types of Asymptotic Notations

1. **Big - Oh (O)**
2. **Big - Omega (Ω)**
3. **Big - Theta (Θ)**

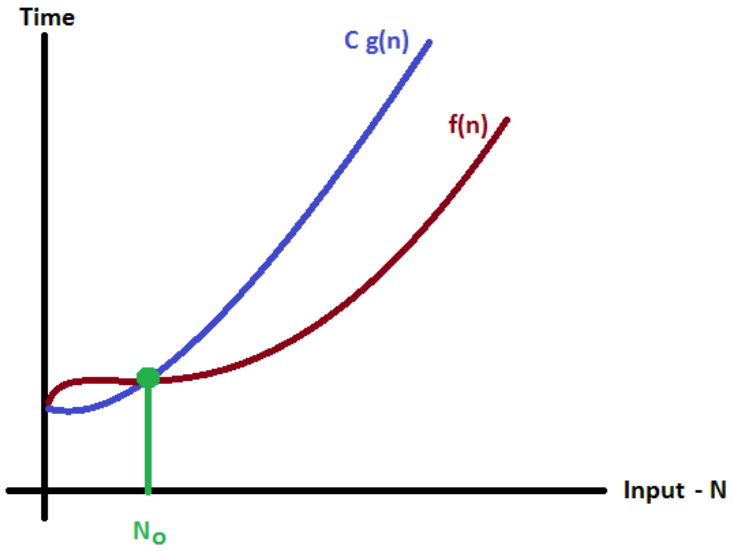
1. Big - Oh Notation (O):

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- That means Big - Oh notation describes the worst case of an algorithm time complexity. Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

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

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and timerequired is on Y-Axis



- In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.
- **Example:**

  Consider the following f(n) and g(n)...

  f(n) = 3n + 2

  g(n) = n

  If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**

  f(n) <= C g(n)

  ⇒3n + 2 <= C n

  Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

  By using Big - Oh notation we can represent the time complexity as follows...

  3n + 2 = O(n)

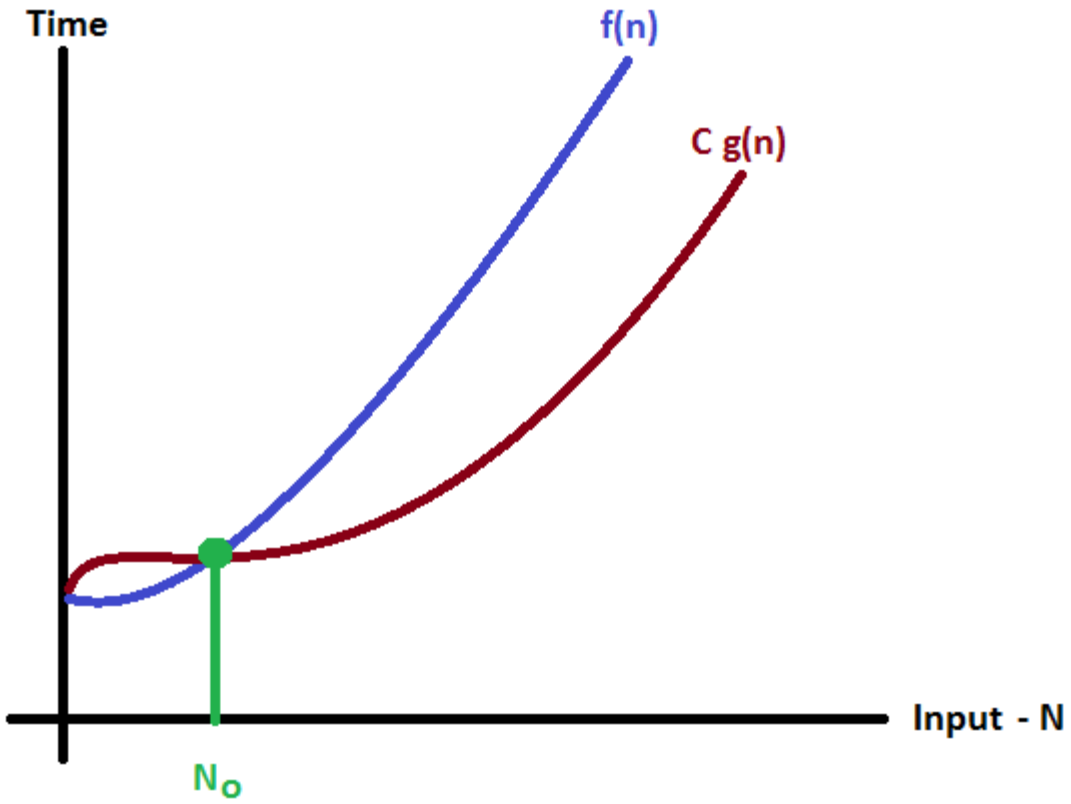
2. Big - Omege Notation (Ω):

- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
- Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and timerequired is on Y-Axis



- In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates thealgorithm's lower bound.
- **Example:**

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 >= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

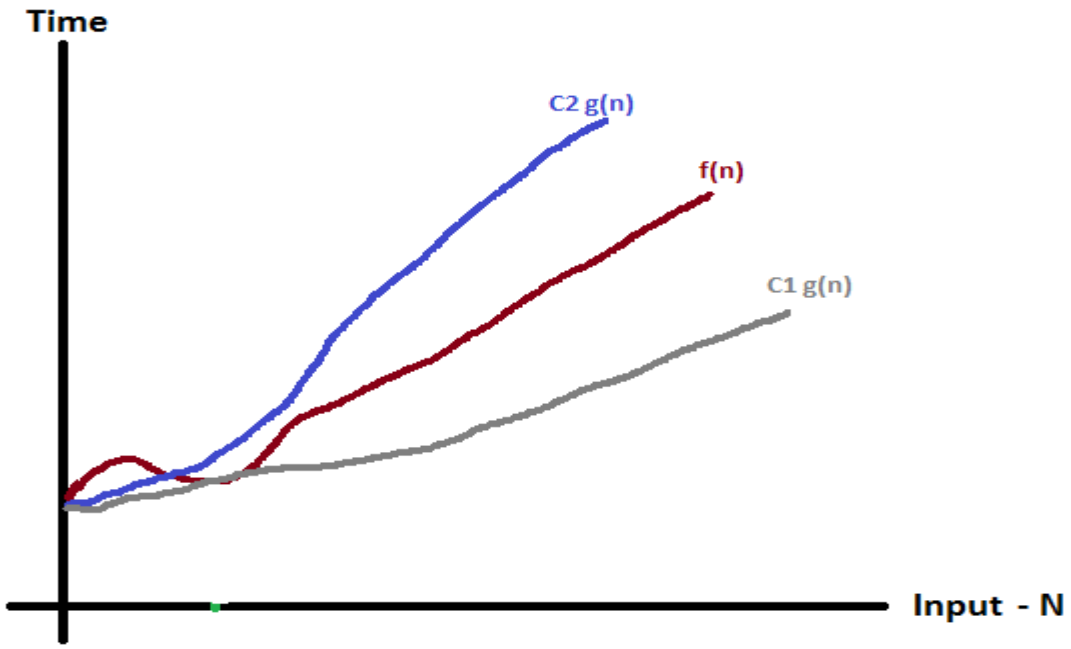
3. Big - Theta Notation (Θ):

- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values.
- That means Big - Theta notation describes the average case of an algorithm time complexity. Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and timerequired is on Y-Axis



- In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater thanf(n) which indicates the algorithm's average bound.

Example:

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**
C_1 g(n) <= f(n) <= C_2 g(n)
⇒C_1 n <= 3n + 2 <= C_2 n
Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.
By using Big - Theta notation we can represent the time compexity as follows...
3n + 2 = Θ(n)

Little oh-o notation:

- A theoretical measure of the execution of an *algorithm*, usually the time or memory needed, given the problem size n, which is usually the number of items.
- Informally, saying some equation f(n) = o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity. The notation is read, "f of n is little oh of g of n".
- **Formal Definition:** f (n) = o (g (n)) means for all c > 0 there exists some k > 0 such that $0 \leq f(n) < cg(n)$ for all $n \geq k$. The value of k must not depend on n, but may depend on c
- As an example, $3n + 4$ is o (n^2) since for any c we can choose $k > (3+ \sqrt{(9+16c)})/2c$. $3n + 4$ is not o (n). o (f(n)) is an upper bound.

Littlee Omega:ω

- A theoretical measure of the execution of an *algorithm*, usually the time or memory needed, given the problem size n, which is usually the number of items.
- Informally, saying some equation f(n) = ω (g(n)) means g(n) becomes insignificant relative to f(n) as n goes to infinity.
- **Formal Definition:** f(n) = ω (g(n)) means that for any positive constant c, there exists a constant k, such that $0 \leq cg(n) < f(n)$ for all $n \geq k$. The value of k must not depend on n, but may depend on c.

1.11. RANDOMIZED ALGORITHMS

Basics of Probability Theory:

- Probability theory has the goal of characterizing the outcomes of natural or conceptual "experiments".
- Examples of such experiments include tossing a coin ten times, rolling a die three times, playing a lottery, gambling, picking a ball form an urn containing white and red balls, and so on.
- Each possible outcome of an experiment is called a sample point and the set of all possible outcomes is known as the sample space S.
- Assume that S is finite (such a sample space is called a discrete sample space).
- An event E is a subset of the sample space S.
- If the sample space consists of n sample points, then there are 2^n possible events.
- **Example:** Tossing three coins when a coin is tossed, there are two possible outcomes: Heads (H) and Tails (T). Consider the experiment of throwing three coins. There are eight possible outcomes: HHH, HHT, HTH, HTT, THH, THT, TTH and TTT. Each such outcome is a sample point. The Sets {HHT, HTT, TTT}, {HHH, TTT} and { } are three possible events.
- The third event has no sample points and is the empty set. For this experiment there are 2^8 possible events.

Definition [Probability] The probability of an event E is defined to be $\frac{|E|}{|S|}$, where S is the sample space.

Ramdomized algorithms are explain in following below topics :

1. Baasic of probabillity theory
2. Randomized Algorithms : An informal Description
3. Identifying the Repeated Element
4. Primality Testing

1.6.1 Basic of Probability Theory

Probability theory is a main conceptual "experiments".

Examples

- ☞ Tossing a coin ten times
- ☞ Rolling a die three times
- ☞ Playing a lottery
- ☞ Gambling
- ☞ Piking
- ☞ A ball from white and red balls etc.,

Sample Point

Each possible outcome of an experiment is called sample point.

Sample space (S)

The set of all possible outcomes is known as the sample space S. S is finite.

Event (E)

An event E is a subset of the S. If the S consists of n sample points, then $2n$ possible events.

1. Probability

The probability of an event E is defined to be $\frac{|E|}{|S|}$, where S is the sample space.

2. Mutual Exclusion

Two events E_1 and E_2 are said to be mutually exclusive. It is not have a common S, i.e., if $E_1 \cap E_2 = \Phi$.

3. Conditional probability

The conditional probability of E_1 given E_2, denoted by prob. $[E_1|E_2]$, is defined as

$$\frac{Prob.[E_1 \cap E_2]}{Prob.[E_2]}$$

4. Independence

Two events E_1 and E_2 said to be idependence.

If prob.$[E_1 \cap E_2]$ = prob.$[E_1]$*prob.$[E_2]$.

5. Random Variable

Let S be the sample of an experiment. A random variable on S is as function that maps the elements of S to the set of real numbers. For any sample point $s \in S$, X(s) denotes the image of s under this mapping. If the range of X, that is, the set of values X can take, is finite, we say X is discrete.

If the sample space of an experiment is $S = \{S_1, S_2, \ldots S_n\}$, the expected value or the mean of any random variable X is defined to be $\sum_{i=1}^{n} Prob.[s_i] * X(s_i) = \frac{1}{n} \sum_{i=1}^{n} X(s_i)$.

7. Probability distribution

Let X be a discrete random variable defined over the sample space S. Let $\{r_1, r_2, \ldots r_n\}$ be its range. Then, the probability distribution of X is the sequence $prob.[X=r_1], prob.[X=r_2] \ldots prob[X=r_m]$. Notice that $\sum_{i=1}^{m} Prob.[X = r_i] = 1$.

8. Binomial distribution

A Bernoulli trial is an experiment, that has two possible outcomes, success and failure.

Let,

success - p

trial time - n

The expected value of X is np, Also,

$$Prob.[X = i] = \binom{n}{i} p^i (1-p)^{n-i}$$

9. Markov's inequality

If X is any nonnegative random variable whose mean is μ, then

$$Prob.[X \geq x] \leq \frac{\mu}{x}$$

Randomized Algorithms : An Informal Description:

- A randomized algorithm is one that makes use of a randomizer (such as a random number generator).
- Some of the decisions made in the algorithm depend on the output of the randomizer.
- The output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run, and the output of a randomized algorithm could also differ from run to run forthe same input.
- The execution time of a randomized algorithm could also vary from run to run for the sameinput.
- Randomized algorithms can be categorized into two classes:
  - ✓ Las Vegas algorithms
  - ✓ Monte Carlo algorithms

Las Vegas Algorithms:

- ❖ The first is algorithms that always produce the same (correct) output for the same input. These are called Las Vegas algorithms.
- ❖ The execution time of a Las vegas algorithm depends on the output of the randomizer.
- ❖ If lucky, the algorithm might terminate fast, and if not it might run for a longer period of time.
- ❖ In general the execution time of a Las Vegas algorithm is characterized as a random variable.

Monte Carlo Algorithms:

- ❖ The second is algorithms whose outputs might differ from run to run (for the same input).
- ❖ These are calledMonte Carlo algorithms.
- ❖ Consider any problem for which there are only two possible answers, say, yes and no.
- ❖ If a Monte Carlo algorithm is employed to solve such a problem then the algorithm might give incorrect answers depending on the output of the randomizer.
- ❖ We require that the probability of an incorrect answer from a Monte Carlo algorithm be low.

1.6.3 Identifying the Repeated Element

Consider an array *a[ ]* of *n* numbers. It has *n/2* elements and *n/2* copies of another element. The problem is to identity the repeated element.

To identify the repeated element to following categories.

(1) Las vegas algorithm

(2) Sampling

Las vegas algorithm takes only $\tilde{O}(\log n)$ time. It randomly picks any array elements and check whether two different cells and value. Then finally we have founded repeated element.

In sampling algorithm, the arts two elements are taken from *n* elements. Thus there is probability that the same array element is picked each time.

Algorithm

Identify the repeated array number .

```
Repeated Element(a,n)
// Finds the repeated element from a [1:n]
{
while (true) do
{
i:=Random( ) mod n+1; j:=Random( ) mod n+1;
// i and j are random number in the range [1,n]
if ((i**j) and (a[i]=a[j])) then return i;
```

}

The run time of the above algorithm is

$$\tilde{O}(\log n)$$

While loop will be identifying repeated number.

i - any one the $\pi/2$ array

i - any one the $\pi/2$ array

Probability - $P = \frac{\pi/2(\pi/2-1)}{\pi^2} > \frac{1}{5}$ for all $n \geq 10$.

1.6.4 Primality Testing

Any integer greater than one is said to be a prime if its only divisors are 1 and the integer itself.

Given an integer n, the problem of deciding whether n is a prime is known as primality testing.

Theorem 1 : [Fermat]

If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$.

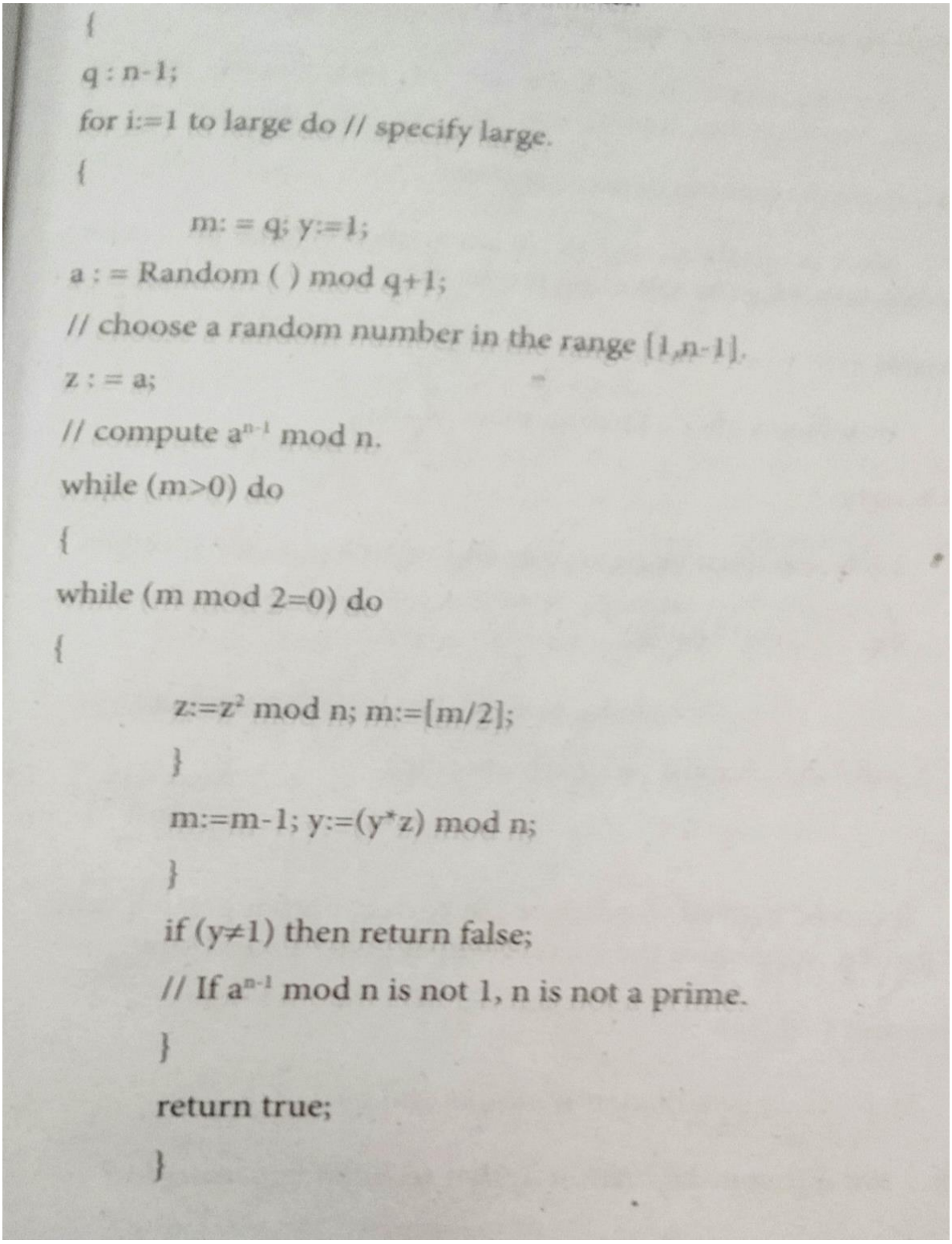
for any integer $a < n$.

Theorem 2 :

The equation $x^2 \equiv 1 \pmod{n}$ has exactly two solutions namely 1 and n-1, if n is prime.

Algorithm :

Prime O(n,α)



```
{
q : n-1;
for i:=1 to large do // specify large.
{
        m: = q; y:=1;
a : = Random ( ) mod q+1;
// choose a random number in the range [1,n-1].
z : = a;
// compute a^(n-1) mod n.
while (m>0) do
{
while (m mod 2=0) do
{
        z:=z^2 mod n; m:=[m/2];
        }
        m:=m-1; y:=(y*z) mod n;
        }
        if (y≠1) then return false;
        // If a^(n-1) mod n is not 1, n is not a prime.
        }
        return true;
        }
```

Advantage and Disadvantage for Randomized Algorithm:

- There are two major advantage of randomize algorithm
  - Those algorithms are simple to implement.
  - These algorithms are many time efficient than traditional algorithm.
- However randomized algorithm may have some drawbacks:
  - The small degree of error may be dangerous for some applications.
  - It is not always possible to obtain better results using randomized algorithm.

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT – II

SEARCH AND SORTING: General Method – Binary Search – Recurrence Equation for Divide and Conquer –Finding the Maximum and Minimum— Merge Sort – Quick Sort – Performance Measurement – Randomized Sorting Algorithm – Selection Sort – A Worst Case Optimal Algorithm – Implementation of Select2 – Stassen's Matrix Multiplications..

2.0 DIVIDE AND CONQUER INTRODUCTION

- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution.
- This mechanism of solving the problem is called the Divide & Conquer Strategy.

- Divide and Conquer algorithm consists of a dispute using the following three steps.

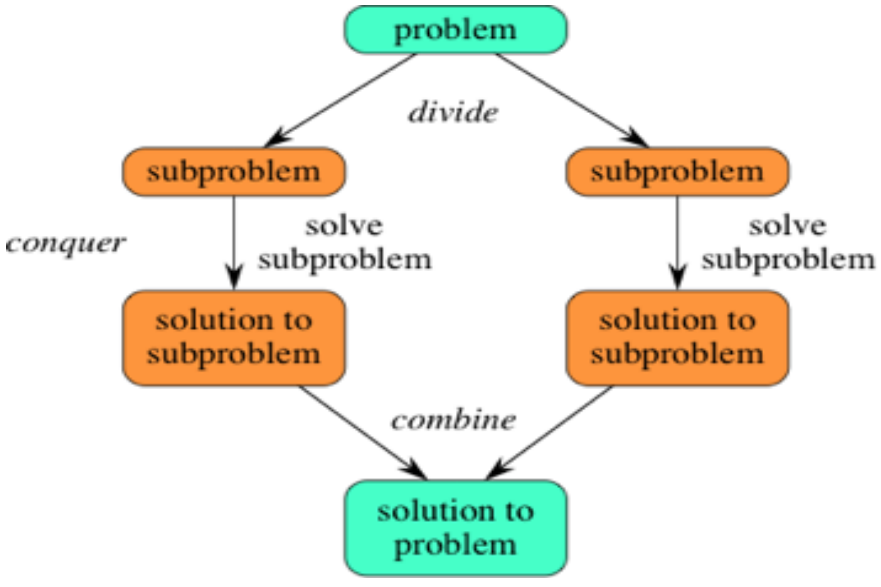
1. **Divide** the original problem into a set of sub problems.
2. **Conquer:** Solve every sub problem individually, recursively.
3. **Combine:** Put together the solutions of the sub problems to get the solution to the whole problem.

2.1 GENERAL METHOD

- In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

- Generally, divide-and-conquer algorithms have three parts –

- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.



- Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not.
- Divide–and–conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

- A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.
- The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
Algorithm DANDC (P)
{
 if SMALL (P) then  return S (p);
 else
{
  divide p into smaller instances p1, p2, …. Pk, 1≥k;
  Apply DANDC to each of these sub problems;
  return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
 }
}
```

- SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting.
- If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DANDC.
- If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Fundamental of Divide & Conquer Strategy

- There are two fundamental 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.

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.

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Advantage of Divide and Conquer Approach

- The difficult problem can be solved easily.
- It divides the entire problem into subproblems
- Efficiently uses cache memory without occupying much space
- Reduces time complexity of the problem
- Solving difficult problems
- Algorithm efficiency
- Parallelism
- Memory access

Disadvantage of Divide and Conquer Approach

- It involves recursion which is sometimes slow
- Efficiency depends on the implementation of logic
- It may crash the system if the recursion is performed rigorously.
- Overhead
- Complexity
- Difficulty of implementation
- Memory limitations
- Suboptimal solutions
- Difficulty in parallelization

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 elementexisting 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.
4. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.

Time Complexity:

- The time complexity of the divide and conquer algorithm to find the maximum and minimum element in an array is O(n).

Space Complexity:

- The space complexity of the divide and conquer algorithm to find the maximum and minimum element in an array is O(log(n)).

2.3 BINARY SEARCH

- If we have 'n' records which have been ordered by keys so that x1 < x2 < … < xn .
- When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search).
- If 'x' is not in the list then j is to set to zero (un successful search).

- In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid].
- If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all.
- Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid].
- If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.
- Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about log 2n times before reaching a trivial length, the worst case complexity of Binary search is about log2n

Algorithm for Binary Search

```
   Algorithm BINSRCH (a, n, x)
  // array a(1 : n) of elements in increasing order, n > 0,
 // determine whether 'x' is present, and if so, set j such that x = a(j)
// else return j
{
  low :=1 ; high :=n ;
  while (low ≤ high) do
 {
  mid :=[(low + high)/2];
  if (x < a [mid]) then high:=mid – 1;
 else if (x > a [mid]) then low:= mid + 1
 else return mid;
}
 return 0;
```

}

- *low* and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one.
- Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons =3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Efficiency of Binary Search

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

- No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.
- There are ten possible ways that an un-successful search may terminate depending upon the value of x.

- If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$$(3+3+3+4+4+3+3+3+4+4)/10=34/10=3.4$$

- The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(logn).

Advantages of Binary Search Technique:

- BST is fast in insertion and deletion when balanced.
- It is fast with a time complexity of O(log n).
- BST is also for fast searching, with a time complexity of O(log n) for most operations.
- BST is efficient.

Complexity:

- The time complexity of the binary search algorithm is O(log n).
- The best-case time complexity would be O(1) when the central index would directly match the desired value. Binary search worst case differs from that.
- The worst-case scenario could be the values at either extremity of the list or values not in the list.

Limitations:

- The recursive method uses stack space.
- Binary search is error-prone.
- Off-by-one errors: While determining the boundary of the next interval, there might be overlapping errors.
- Caching is poor.

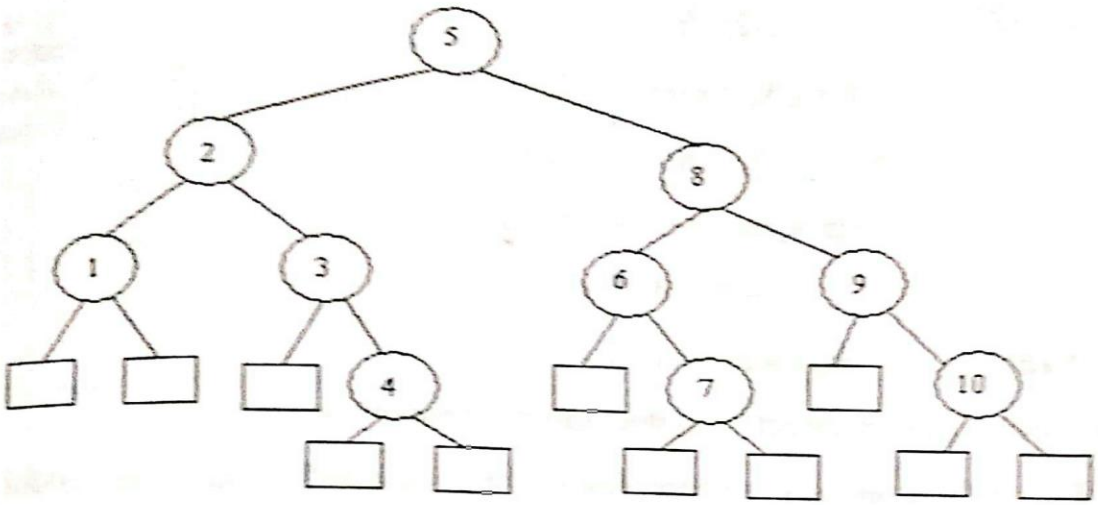
Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

Space Complexity

Space Complexity	O(1)

The search tree for the above list of elements will be as given below.



If the element is present, then search end in a circular node (inner node), if the element is not present, then it end up in a square node (leaf node)

Now we will consider the worst case, average case and best case complexities of the algorithm for a successful and an unsuccessful search.

Worst Case

Find out k, such that $2^{k-1} <= n <= 2^k$

Then for a successful search, it will end up in either of the k inner nodes. Hence the complexity is O(k), which is equal to $O(\log_2 n)$.

For an unsuccessful search, it need either k-1 or k comparisons. Hence complexity is Θ (k), which is equal to $\Theta(\log_2 n)$.

Average Case

Let I and E represent the sum of distance of all internal nodes from root and sum of distance of all external nodes from the root respectively. Then

$E = I + 2n$

Let $A_s(n)$ and $A_u(n)$ represents average case complexity of a successful and unsuccessful search respectively. Then

$A_s(n) = 1 + I/n$

$A_u(n) = E / (n+1)$

$A_s(n) = 1 + I/n$

$= 1 + (E-2n)/n$

$= 1 + (A_u(n)(n+1) - 2n)/n$

$= (n + (A_u(n)(n+1) - 2n))/n$

$= (n(A_u(n) -1) + A_u(n))/n$

$= A_u(n) - 1 + A_u(n)/n$

$A_s(n) = A_u(n)(1+1/n) - 1$

$A_s(n)$ and $A_u(n)$ are directly related and are proportional to $\log_2 n$.

Hence the average case complexity for a successful and unsuccessful search is $O(\log_2 n)$

Best Case

Best case for a successful search is when there is only one comparison that is at middle position if there is more than one element. Hence complexity is $\Theta(1)$

Best case for an unsuccessful search is $O(\log_2 n)$

	Best Case	Average Case	Worse Case
Successful	$\Theta(1)$	$O(\log_2 n)$	$O(\log_2 n)$
Unsuccessful	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

2.6.3 Advantages and Disadvantages of binary search

Advantages

- ☞ In this method elements are eliminated by half each time. So it is very faster than the sequential search.
- ☞ It requires less number of comparisons than sequential search to locate the search key element.

Disadvantages

- ☞ An insertion and deletion of a record requires many records in the existing table be physically moved in order to maintain the records in sequential order.
- ☞ The ratio between insertion/deletion and search time is very high.

Table : Difference between sequential and binary search

Sequential technique	Binary search technique
This is the simple technique of searching an element	This is the efficient technique of searching an element
This technique does not require the list to be sorted	This technique require the list to be sorted. Then only this method is applicable
The worst case time complexity of this technique is $O(n)$	The worst case time complexity of this technique is $O(\log n)$
Every element of the list may get compared with the key element.	Only the mid element of the list is compared with key element.

2.2 FINDING THE MAXIMUM AND MINIMUM

1. Let us consider simple problem that can be solved by the divide and conquer technique.

2. The problem is to find the maximum and minimum value is a set of *'n'* elements.

3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.

4. Hence, the time is determined mainly by the total cost of the element comparison.

Algorithm

Algorithm *StraightMaxMin(a,n, max,min)*

```
// Set max to the maximum and min to the minimum of a[1:n].
{
max := min:= a[1];
for i:=2 to n do
{
if (a[i]>max) then max : = a[i];
if (a[i]<min) then min:=a[i];
}
}
```

Explanation

a. Straight maximum requires $2(n-1)$ element comparisons in the best, average,worst causes.

b. By realizing the comparison of $a[i]$ max is false, improvement in a algorithm can be done.

c. Hence we can replace the constants of the for loop by, if $(a[i]>max)$ then $max = a[i]$; Else if $(a[i]<2(n-1))$

d. On the average $a[i]$ is > max half the time, and so the average no. of comparison is $3n/2-1$.

A Divide and conquer algorithm for this problem would proceeds as follows :

a. Let $P=(n, a[i],...a[j])$ denote an arbitrary instance of the problem.

b. Here 'n' is the no.of elements in the list $(a(i),....a(j))$ and we are interested in finding the maximum and minimum of the list.

c. If the list has more than 2 elements, p has to be divided into smaller instances.

d. For example, we might divide 'P' into the 2 instances, $P_1 = ([n/2],a[1],...a[n/2])$ & $P_2 = (n-(n/2), a((n/2)+1),....a(n))$. After having divided '$P$' into 2 smaller sub problems, we can solve then by recursively invoking the same divide and conquer algorithm.

Algorithm :

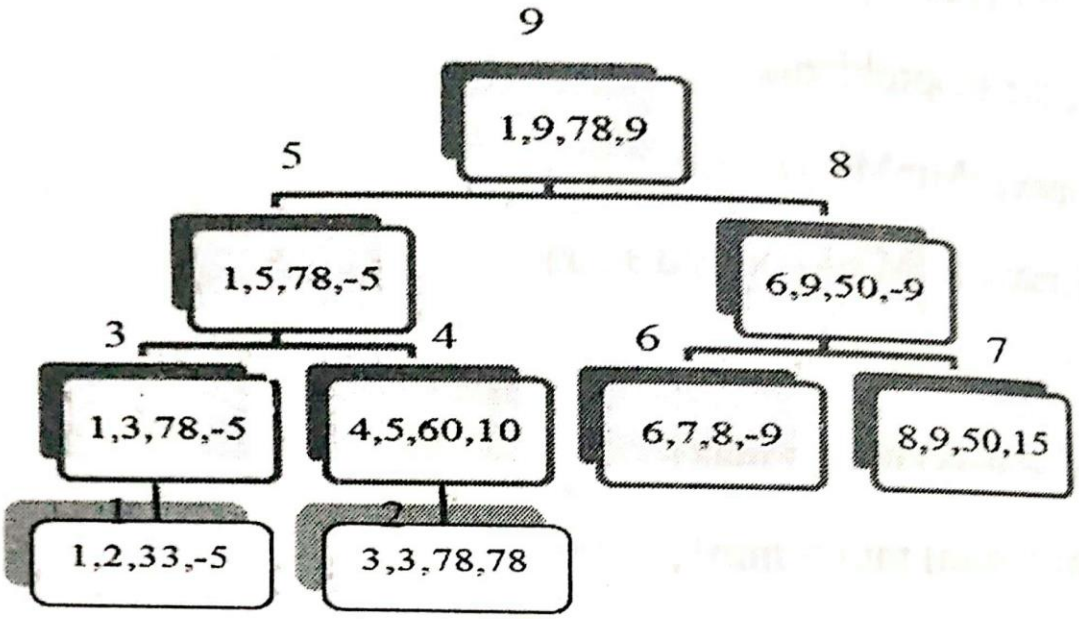
```
Algorithm MaxMin(i,j, max, min)
// a[1:n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
// largest and smallest values in a[i:j], respectively.
{
if(i=j) then max:=min:=a[i]; // Small(P)
else if (i=j-1) then // Another case of Small(P)
{
if (a[i]<a[j]) then
{
max:=a[j];min:=a[i];
}
else
{
max:=a[i];min:=a[j];
```

```
}
}
else
{
// If P is not small, divide P into subprograms.
// Find where to split the set.
mid:=[(i+j)/2];
//Solve the subprograms.
MaxMin (i,mid, max, min1);
MaxMin (mid+1,j,max1,min1);
//Combine the solutions.
if(max<max1) then max:=max1;
if(min>min1)then min:=min1;
}
}
```

MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements { *a(i), a(i+1),....,a(j)*}. The situation of set sizes one (*i=j*) and two (*i=j-1*) are handled separately. For sets containing more than two elements, the midpoint is determined and two new subprograms are generated. When the maxima and minima of these subprograms are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

Example of Finding the Maximum and Minimum

Tree Diagram



i. As shown in figure, in this Algorithm, each node has 4 items of information: i, r, max and min.

ii. In figure, root node contains 1 and 9 as the values of i and r corresponding to the initial call to MaxMin.

iii. This execution produces 2 new calls to MaxMin, where i and r have the values 1, 5 and 6, 9 respectively and thus split the set into 2 subsets of approximately the same size.

iv. Maximum depth of recursion is 4.

Efficiency of Finding the Maximum and Minimum

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n)=\begin{cases} T([n/2])+T([n/2])+2 & n>2 \\ 1 & n=2 \\ 0 & n=1 \end{cases}$$

When n is a power of two, $n=2k$ for some positive integer k, then

$T(n) = 2T(n/2)+2$

$= 2(2T(n/4)+2)+2$

$= 4T(n/4)+4+2$

$= 2^{k-1} T(2) +\sum_{1<i<k-1} 2^i$

$= 2^{k-1} + 2^k-2 = 3n/2-2\backslash 1$

Note that $3n/2-2$ is the best, average, and worst-case number of comparisons when n is a power of two.

Advantages:

- It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI.
- ***Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game.***

Limitations

- Because of the huge branching factor, the process of reaching the goal is slower.
- Evaluation and search of all possible nodes and branches degrades the performance and efficiency of the engine.
- Both the players have too many choices to decide from.

Complexity

- Time complexity- Min-Max algorithm is O(bm), where b is branching factor of the game-tree, and m is the maximum depth of the tree. i.e **O(n)**
- Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

2.4 Merge Sort

- This is a simple and very efficient algorithm for sorting a list of numbers, called MergeSort. We are given an sequence of n numbers A, which we will assume is stored in an array A[1 ...n].
- The objective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

The major elements of the Merge Sort algorithm.

- ✓ **Divide:** Split A down the middle into two subsequences, each of size roughly n/2.
- ✓ **Conquer:** Sort each subsequence (by calling MergeSort recursively on each).
- ✓ **Combine:** Merge the two sorted subsequences into a single sorted list.

- The dividing process ends when we have split the subsequences down to a single item. A sequence of length one is trivially sorted.
- The key operation where all the work is done is in the combine stage, which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

In merge sort we follow the following steps:

- We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
- Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.
- Then we divide these 2 subarrays again, just like we divided our main array and this continues.
- Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Algorithm for Merge Sort

```
Algorithm MergeSort(array A, int p, int r)
{
  if (p < r)
  {                                  // we have at least 2 items
    q := (p + r)/2 ;
     MergeSort(A, p, q)  ;      // sort A[p..q]
     MergeSort(A, q+1, r);   // sort A[q+1..r]
     Merge(A, p, q, r);        // merge everything together
  }
 }
```

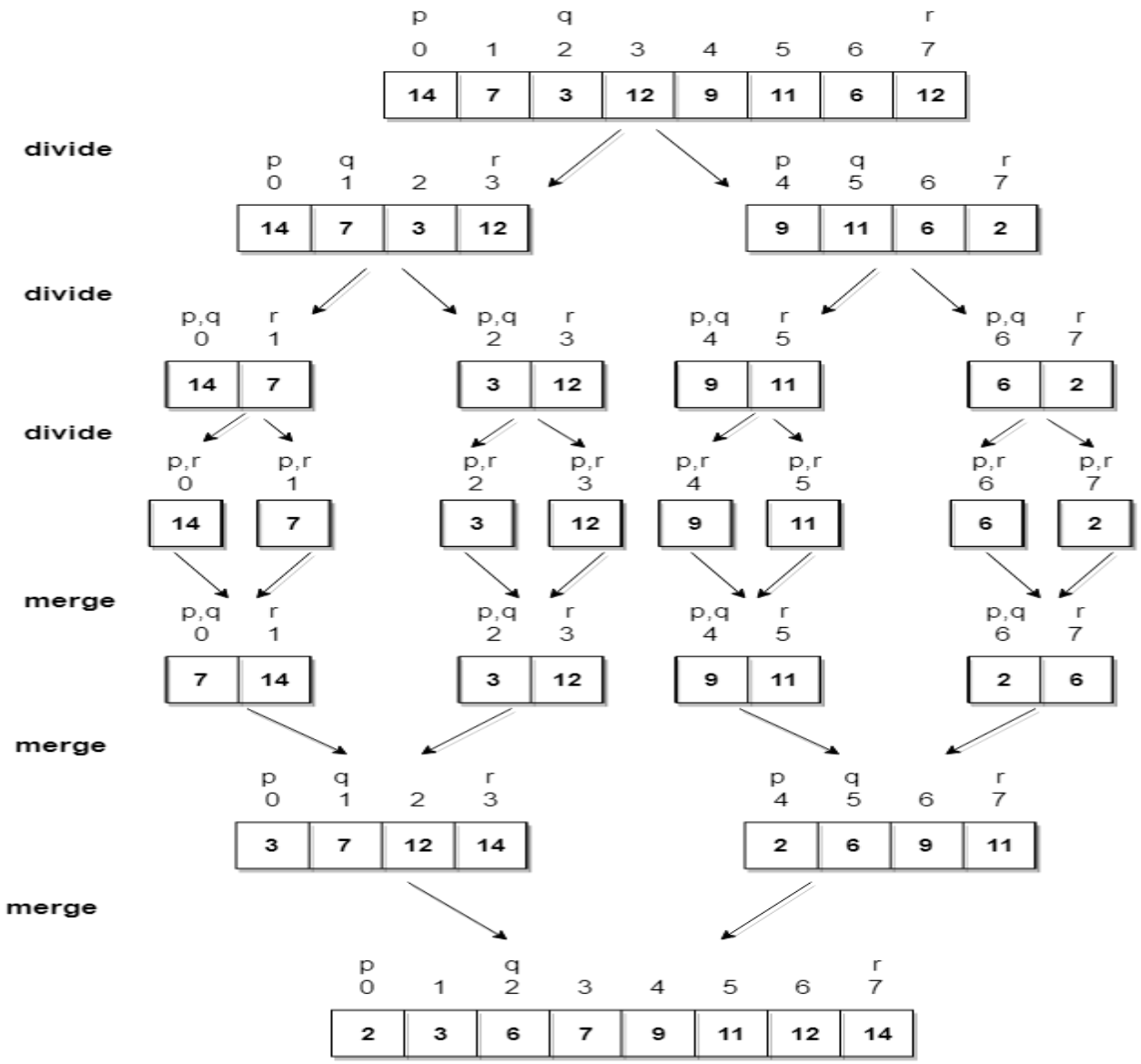
Algorithm for Merging 2 Sorted sub arrays

```
Algorithm Merge(array A, int p, int q, int r)
{                                        // merges A[p..q] with A[q+1..r]
  array B[p..r];
   i = k = p;                          // initialize pointers
  j = q+1;
  while (i <= q and j <= r)
  {                                    // while both subarrays are nonempty
   if (A[i] <= A[j])
```

```
  B[k++] = A[i                    // copy from left subarray
 else
 B[k++] = A[j                   // copy from right subarray
}
 while (i <= q)
B[k++]:= A[i++];        // copy any leftover to B
while (j <= r)
B[k++] := A[j++];
for i = p to r do
A[i] := B[i] ;           // copy B back to
}
```

Example for Merge Sort

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}
Below, we have a pictorial representation of how merge sort will sort the given array.



Representing this in O notation:

T(n) = **O(n log n)**

- We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.
- Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. ***The Best and worst case time complexity of Merge sort is O(n log n)***.

Time Complexity of merge sort

Best Case	Average Case	Worst Case
O(n log n)	O(n log n)	O(n log n)

Advantage for Merge Sort

- It is quicker for larger lists because unlike insertion and bubble sort it doesnt go through the whole list seveal times.
- It has a consistent running time, carries out different bits with similar times in a stage.

Disadvantage for Merge Sort

- Slower comparative to the other sort algorithms for smaller tasks.
- Goes through the whole process even i he list is sorted (just like insertion and bubble sort?)
- Uses more memory space to store the sub elements of the initial split list.

Complexity Analysis of Merge Sort:

- Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
- $T(n) = 2T(n/2) + \theta(n)$
- Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Advantages of Merge Sort:

- Stability: Merge sort is a stable sorting algorithm,
- Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

- Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data.
- Not always optimal for small datasets: For small datasets, Merge sort has a higher time complexity than some other sorting algorithms

Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Time Complexity

- Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation. **T(n) = 2T(n/2) + O(n)** The solution of the above recurrence is O(nLogn)

Space Complexity:

- The space complexity of **Merge sort is O(n).**
- This means that this algorithm takes a lot of space and may slower down operations for the last data sets.

2.5 QUICK SORT

2.6 Quick Sort

Quick sort also known as "partition-exchange sort". It is one of the widely used internal sorting algorithms. In its basic form it was developed by C.A.R Hoare in 1960. Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is O (n log n).

Divide and conquer strategy for quick sort to sort an array A [p...r] is as follows:

Divide: partition the array A [p...r] into two sub-arrays A[p...q -1] and A [q + 1.....r] such that each element of A [p...q -1] is less than or equal to A [q], which in turn, less than or equal to each element of A [+ 1...r]. Compute the index q as part of this partitioning procedure.

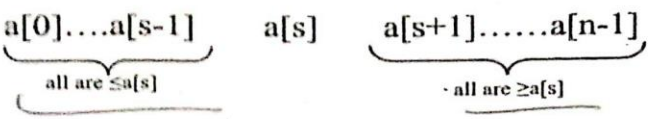
Conquer: Sort the two subarrays A[p…q-1] and A[q+1…r] by recursive calls to quick sort .

Combine: Since the subarrays are sorted in place, no work is needed to combine them. The entire array A [p…r] is now sorted.

Partition

It is a situation where all the elements before the position 's' are smaller than or equal to *a[s]* and all the elements after position 's' are greater than or equal to *a[s]*.

The partition is shown as



After partitioning, *a[s]* will be in its final position in the sorted array.

Then sorting of element of two sub arrays preceding and following *a[s]* can be done independently.

Note the difference with merge sort: there, the division of the problem into two sub problems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

Partition of all 'small' elements	Partition of all 'large' elements
Sub array 1	Sub array 2

Here is pseudocode of quick sort: call Quick sort(A[0..$n-1$])

2.4.1 Algorithm for Quick Sort

ALGORITHM Quicksort(A[l..r])

//Sorts a subarray by quicksort

//Input: Subarray of array A[0..$n-1$], defined by its left and right

// indices l and r

//Output: Subarray A[l..r] sorted in nondecreasing order

if l<r

$s \leftarrow$Partition(A[l..r]) //s is a split position

Quicksort(A[l..$s-1$])

Quicksort(A[$s+1$..r])

Alternatively, we can partition A[0..$n-1$] and, more generally, its sub array A[l..r] ($0 \le l < r \le n-1$) by the more sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quick sort.

As before, we start by selecting a pivot—an element with respect to whose value we are going to divide the sub array. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the sub array's first element: $p = A[l]$.

Unlike the Lomuto algorithm, we will now scan the sub array from both ends, comparing the sub array's elements to the pivot. The left-to-right scan, denoted below by index pointer i, starts with the second element.

Since we want elements smaller than the pivot to be in the left part of the sub array, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index pointer j, starts with the last element of the sub array.

Since we want elements larger than the pivot to be in the right part of the sub array, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot. Why is it worth stopping the scans after encountering an element equal to the pivot? Because doing this tends to yield more even splits for arrays with a lot of duplicates, which makes the algorithm run faster. For example, if we did otherwise for an array of n equal elements, we would have gotten a split into sub arrays of sizes $n-1$ and 0, reducing the problem size just by 1 after scanning the entire array.

ALGORITHM HoarePartition(A[l..r])

//Partitions a subarray by Hoare's algorithm, using the first element

// as a pivot

//Input: Subarray of array A[0..$n-1$], defined by its left and right

// indices l and r (l < r)

//Output: Partition of A[l..r], with the split position returned as

// this function's value

$p \leftarrow$ A[l]

$i \leftarrow$ l; $j \leftarrow r+1$

repeat

repeat $i \leftarrow i+1$ until A[i] $\geq p$

repeat $j \leftarrow j-1$ until A[j] $\leq p$

swap(A[i], A[j])

until $i \geq j$

swap(A[i], A[j]) //undo last swap when $i \geq j$

swap(A[l], A[j])

return j

Analysis

Here the basic operation is comparison. The number of key comparison before a partition is achieved is n+1.

Best case

If all splits happen in the middle of the subarrays, it is the best case.

$$C_{best}(n) = 2\,C_{best}(n/2) + n \quad \text{for } n>1.$$

$$\& \quad C_{best}(1) = 0$$

According to master theorem, $C_{best}(n) \in O(n \log_2 n)$. When solving it for $n=2k$, $C_{best}(n) = n \log 2n$.

Worst case

If the list is in descending order it is the worst case. The total number of key comparisons can be given as

$$C_{worst}(n) = (n+1) + n + \ldots\ldots + 3 = (n+1)(n+2) - 3 \in \theta(n^2)$$

$$C_{worst}(n) \in \theta$$

Average case:

If the array is randomly ordered it is the average case. The total number of key comparison made by quick sort in average case is given as,

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \text{ for } n > 1,$$

$$\& \; C_{avg}(0) = 0, \quad C_{avg}(1) = 1$$

Therefore $C_{avg}(n) = 2n \ln n \approx 1.38\, n \log 2n$

Importance of quicksort

- ☞ In average case quicksort, makes only 38% more comparison than the best case quicksort.
- ☞ Its innermost loop is so efficient & so it runs faster than mergesort on randomly ordered arrays.

- ☞ Improvements can be made in quick sort algorithm by using better pivot selection method.

Example 1 :

Example for Quick Sort

Explain with Example (1)

The array elements are

5 3 1 9 8 2 4 7

∴ P – pivot

P=5 i=3 j=7

3 5 7 5
i<P condition true j>P condition true so it i value is increment and j value is decrement.

P=5 i=1 j=4

1 5 4 5
i<P condition true j>P condition false so it is i value increment and j value not decrement.

P i j

5	3	1	9	8	2	4	7

i<P condition false j>P condition false so it is

Now exchange the 9 and 4

P i j

5	3	1	4	8	2	9	7

i<P condition true j>P condition true i incremented

j decremented

P i j

5	3	1	4	8	2	9	7

i<P condition false j>P condition false so now exchang

the 8 and 2.

P i j

5	3	1	4	2	8	9	7

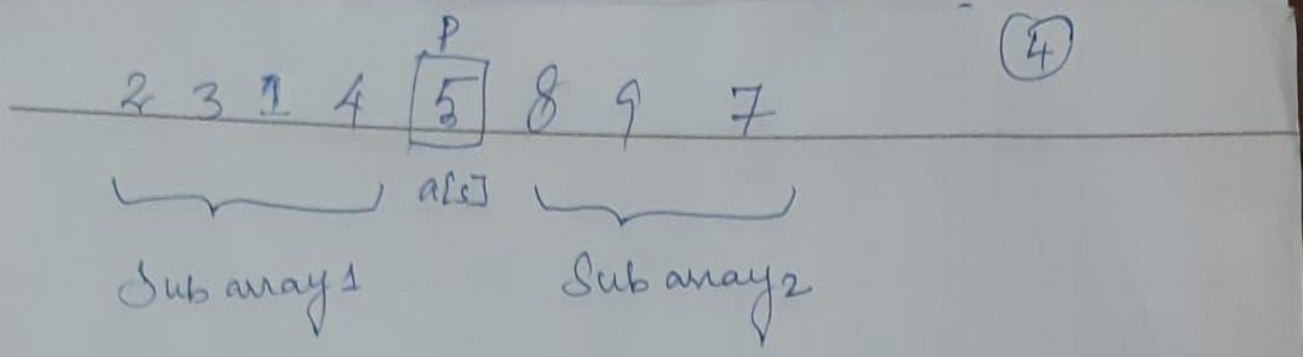
i<P condition true j>P condition true so i - incremented j - decremented

P			j	i			
5	3	1	4	2	8	9	7

i<P condition false j>P condition false but not possible exchange because a[i]<P condition true so swap the pivot value and j value

				P			
2	3	1	4	5	8	9	7

Now the array has been sub divided into sub array with pivot element as middle.



take Sub array 1

P	i		j
2	3	1	4

$i < p$ condition false $j > p$ condition true so i not increment

j decrement

P	i	j
2	3	1 4

$i < p$ condition false $j > p$ condition false both are condition false Exchange the i and j value that is 3 and 1

P	i	j	
2	1	3	4

1 < 2 3 2

i < p condition true j > p condition true both condition true

i incremented and j decremented.

p	j	i	
2	1	3	4

3 1 2

i < p condition false j ≥ p condition false Both condition false But

a[j] < p condition is true Swap p and j that is 2 and 1

	p		
1	2	3	4

Subarray3

Subarray3

p	ij
3	4

Here i = j i.e, both points to the same element. therefor allelement sorted in ~~same~~ subarray1. and subarray3.

Subarray 2

p i j

[8] 9 7

i<p condition false (9 8) j>p condition false (7 8) both condition false so Exchange i and j value that is 9 and 7

p i j

[8] 7 9

i<p condition true (7 8) j>p condition true (9 8) both condition true i increment and j decrement.

j i

[8] 7 9

i<p condition false (9 8) j>p condition false (7 8) a[i]<p condition true swap 8 and 7 that is pivot and j.

p

7 [8] 9

Hence the array elements are sorted The sorted array is

1 2 3 4 5 7 8 9.

2.4.2 Efficiency of Quick Sort

The number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices i and j cross over.

The number of key comparisons is n, if the scanning indices i and j coincides.

1. Best Case Analysis(split in the middle)

If the array is always partitioned at the *mid* , then it brings the best case efficiency of an algorithm

The number of key comparisons in the best case satisfies the recurrence

$C_{best}(n) = 2\, C_{best}(n/2) + n$ for $n > 1$,

(Or)

$$C(n) = C(n/2) + C(n/2) + n \quad \text{----------(1)}$$

Time required to sort left sub array

Time required to sort right sub array

Time required for partitioning the sub array

and $C_{best}(1) = 0$.

(i) Using Master Theorem

Solve equation (1) using Master Theorem

If $f(n) \in \Theta(n^d)$ then

$T(n) = \Theta(n^d)$ if $a < b^d$

$T(n) = \Theta(n^d \log n)$ if $a = b^d$

$T(n) = \Theta(n \log_b^a)$ if $a > b^d$

$$C(n) = 2\,C(n/2) + n$$

Here, f(n) $\in n^1$ therefore d = 1

Now , $a = 2$ and $b = 2$

As from case 2 we get $a = bd$ i.e. 2 = 21

We get ,

$T(n)$ i.e $C(n) = \Theta(n^d \log n)$

$C_{best}(n) = \Theta(n \log n)$

Best case time complexity of quick sort is $\Theta(n \log n)$

(ii) Using substitution method

$C(n) = C(n/2) + C(n/2) + n$ ----------(1)

$$C(n) = 2C(n/2) + n$$

Assume $n = 2^K$ since each time the list is divide into two equal halves then equation becomes,

$C(2^K) = 2C(2^k/2) + 2^k$

$C(2^K) = 2C(2^{k-1}) + 2^k$

Now substitute $C(2^{k-1}) = 2C(2^{k-2}) + 2^{k-1}$

$C(2^K) = 2[2C(2^{k-2}) + 2^{k-1}] + 2^k$

$C(2^K) = 2^2C(2^{k-2}) + 2.2^{k-1} + 2^k$

$C(2^K) = 2^2C(2^{k-2}) + 2^k + 2^k$

$C(2^K) = 2^2C(2^{k-2}) + 2.2^k$

If we substitute $C(2^{k-2})$ then ,

$C(2^K) = 2^2C(2^{k-2}) + 2.2^k$

$C(2^K) = 2^2[2\,C(2^{k-3}) + 2^{k-2}] + 2.2^k$

$C(2^K) = 2^3C(2^{k-3}) + 2^2.2^{k-2} + 2.2^k$

$C(2^K) = 2^3C(2^{k-3}) + 2^k + 2.2^k$

$C(2^K) = 2^3C(2^{k-3}) + 3.2^k$

Similarly we can write

$C(2^K) = 2^4C(2^{k-4}) + 4.2^k$

$C(2^K) = 2^kC(2^{k-k}) + k.2^k$

$C(2^K) = 2^kC(20) + k.2^k$

$C(2^K) = 2^kC(1) + k.2^k$

$C(1) = 0$

Hence the above equation becomes

$C(2^K) = 2^k.0 + k.2^k$

now as we assumed $n = 2^k$ we can also say

$n = \log_2 n$ [by taking logarithm on both side]

$$C(n) = n.0 + \log_2 n \,.\, n$$

Thus it is proved that best case time complexity of quick sort is $\Theta(n \log n)$

2. Worst Case Analysis (sorted array)

The worst case for quick sort occurs when the pivot is a minimum or maximum of all the elements in the list .

For example,

if A[0..$n-1$] is a strictly increasing array and we use A[0] as the pivot,

The left-to-right scan will stop on A[1]

The right-to-left scan continues upto A[0]

The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n-1) + n$$

$$C_{worst}(n) = (n-1) + (n-2) + \ldots + 2 + 1$$

But as we know

$$1 + 2 + 3 + \text{----} + n = n(n+1)/2 = \tfrac{1}{2} n^2$$

$$C_{worst}(n) \in \Theta(n^2)$$

The time complexity of worst case of quick sort is $\Theta(n^2)$

3. Average Case Analysis (random array)

Let $C_{avg}(n)$ be the average number of key comparison made by Quick Sort.

The partition split can be happen in each position S ($0 \le S \le n-1$) with the probability $1/n$.

The recurrence relation is

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \text{ for } n > 1,$$

$$C_{avg}(0) = 0,$$

$$C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n \approx 1.39 n \log_2 n.$$

Thus, on the average case, Quick Sort makes 38% more comparison the best case.

Hence runs faster than mergesort on randomly ordered analys.

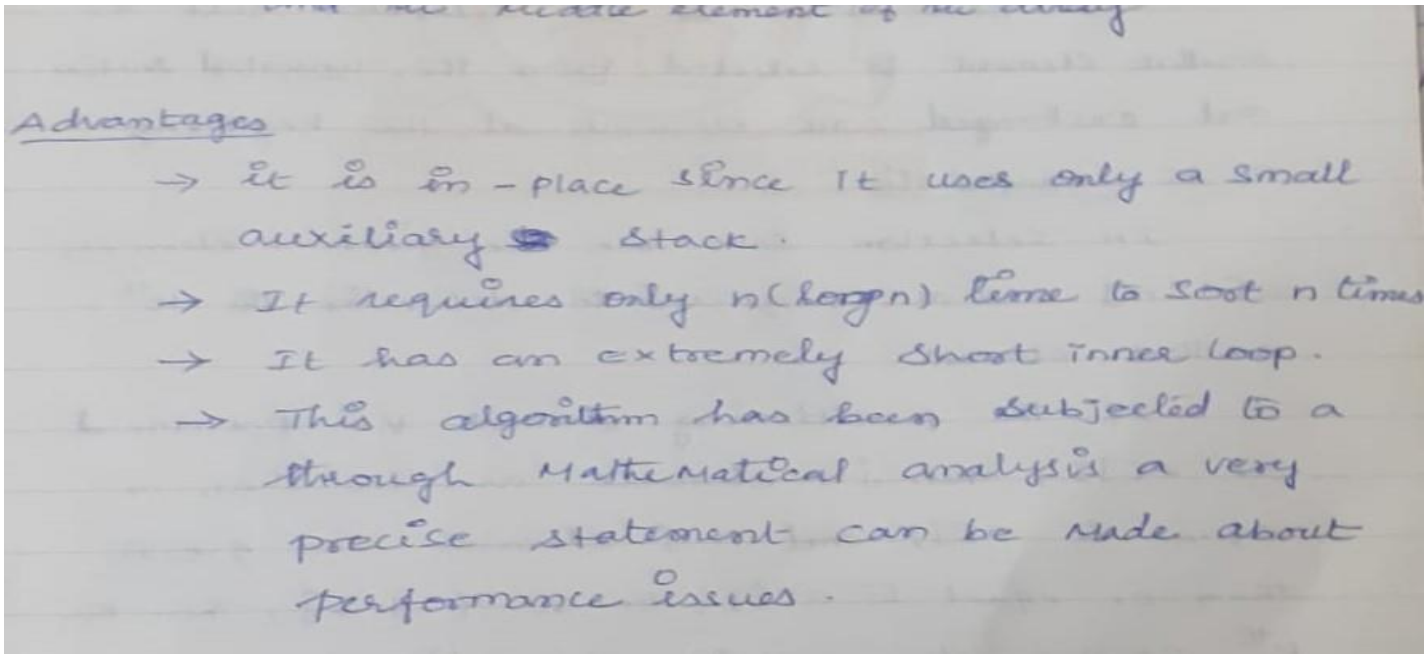
Time complexity of quick sort

Best case	*Average case*	*Worst case*
$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

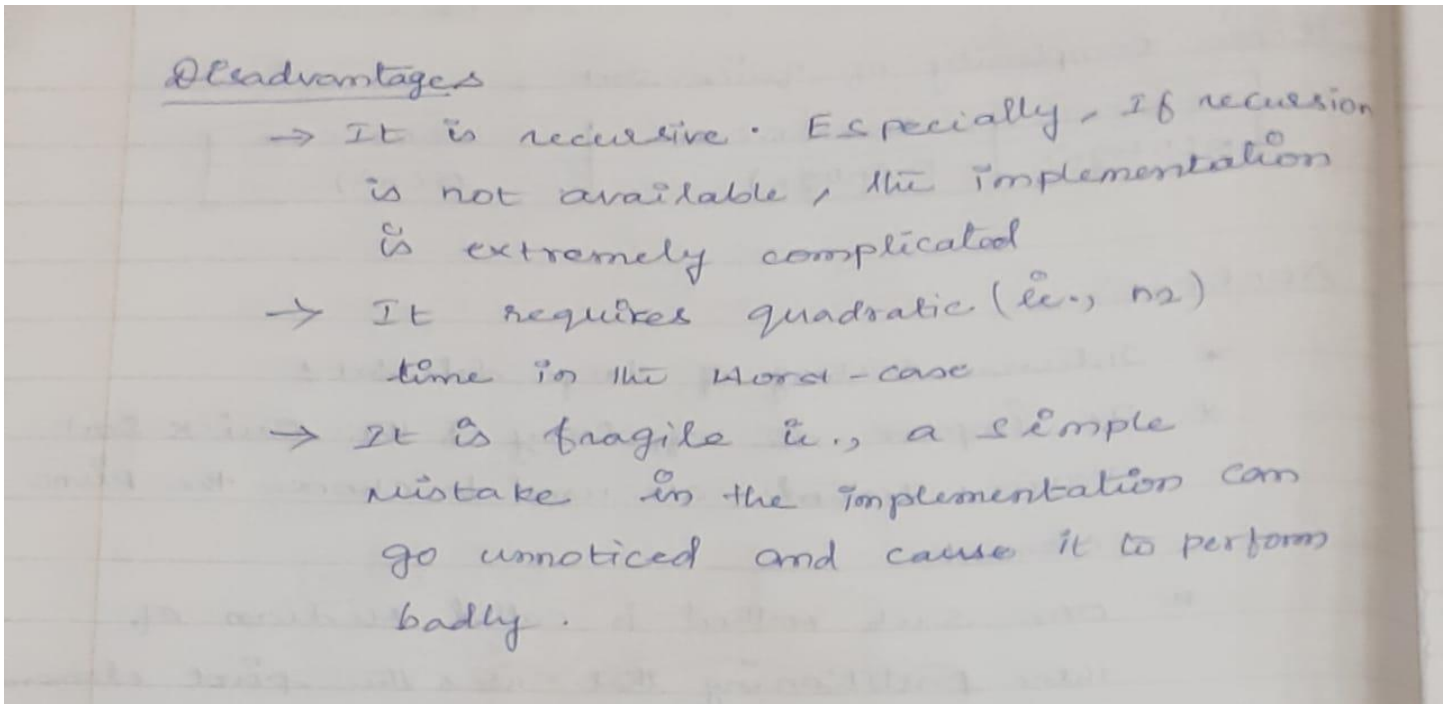
Applications

- ☞ Internal sorting of large data sets.
- ☞ To improve the efficiency of the Quick sort various methods are used to choose the pivot element.
- ☞ One such method is called, median of three partitioning that uses the pivot element as the median of left most, right most and the middle element of the array.

Advantage of Quick Sort



Disadvantage of Quick Sort



Randomized Sorting Algorithm: (Random quick sort)

While sorting the array a[p:q] instead of picking a[m], pick a random element (from among a[p], a[p+1], a[p+2]---a[q]) as the partition elements.

The resultant randomized algorithm works on any input and runs in an expected O(n log n) times.

Algorithm for Random Quick sort

```
Algorithm RquickSort (a, p, q)
{
If(high>low) then
{
If((q-p)>5) then
Interchange(a, Random() mod (q-p+1)+p, p);
 m=partition(a,p, q+1);
quick(a, p, m-1);
quick(a,m+1,q);
}
}
```

Comparison between Merge and Quick Sort

- Both follows Divide and Conquer rule.
- Statistically both merge sort and quick sort have the same average case time i.e., O(n log n).
- Merge Sort Requires additional memory. The pros of merge sort are: it is a stable sort, and there is no worst case (means average case and worst case time complexity is same).
- Quick sort is often implemented in place thus saving the performance and memory by not creating extra storage space.
- But in Quick sort, the performance falls on already sorted/almost sorted list if the pivot is not randomized. Thus why the worst case time is $O(n^2)$.

Randomized Quicksort works:

- **Choose Pivot** Randomly select an element from the array to serve as the pivot element
- **Partitioning:** Rearrange the elements in the array so that all elements less than the pivot are to its left and all elements greater than the pivot are to its right. The pivot is now in its final sorted position.
- **Recursive sorting:** Recursively apply the same process to the sub arrays on the left and right of the pivot
- **Combine:** Once all subarrays are sorted the entire array will be sorted.

Uses of Random Algorithm:

- ✓ Randomized algorithms have various applications in computer science and mathematics, such as ***cryptography, data structures, machine learning, optimization, and computational geometry.***

Need for Randomized Algorithm

- ✓ Improved Efficiency:
- ✓ Complex Problems can be Handled Effectively:
- ✓ Worst-Case Scenarios can be avoided
- ✓ Cryptography

2.5 SELECTION SORT

In selection sort, we are given n elements $a[1:n]$ and are required to determine the k^{th}- smallest element.

If the partitioning element V is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k>j$, then the k^{th}-smallest element is in $a(1:j-1)$.

If $k=j$, then $a(j)$ is the k^{th}-smallest element, and if $k>j$, then the k^{th}- smallest element is the $(k-j)^{th}$ - smallest element in $a[j+1:n]$.

Algorithm

Algorithm Select1(a,n,k)

```
// Selects the k^th smallest element in a[1:n] and places it
// in the k^th position of a[ ]. The remaining elements are
// rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
// a[m] ≥ a[k] for k < m < n.
{
low : =1; up:=n+1;
a[n+1] := ∞; //a[n+1] is set to infinity.
repeat
{
// Each time the loop is entered,
// 1 ≤ low ≤ k ≤ up ≤ n+1.
j:=Partition (a,low,up);
// j is such that a[j] is the j^th-smallest value in a[ ].
if (k=j) then return;
else if (k<j) then up:=j; // j is the new upper limit.
else low := j+1; // j+1 is the new lower limit.
}until (false)
}
```

The reulting algorithm is function Select1. This function places the k^{th}-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \le a[k]$, $1 \le i < k$, and $a[i] \ge a[k]$, $k < i \le n$.

The following steps constitute the Selection Sort algorithm:

Step 1: Create a variable MAX to store the maximum of the values scanned upto a particular stage. Also create another variable say MAX-POS which keeps track of the position of such maximum values.

Step 2: In each iteration, the whole list/array under consideration is scanned once to find out the current maximum value through the variable MAX and to find out the position of the current maximum through MAX-POS.

Step 3: At the end of an iteration, the value in last position in the current array and the (maximum) value in the position MAX-POS are exchanged.

Step 4: For further consideration, replace the list L by L ~ {MAX} {*and the array A by the corresponding subarray*} and go to Step 1.

Example for Selection Sort

Example 1:

Unsorted list:	5	2	1	4	3

1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

Swap 5 and 1	1	2	5	4	3

2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

No Swap	1	2	5	4	3

3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3	1	2	3	4	5

4th iteration:

Smallest = 4

4 < 5, smallest = 4

No Swap	1	2	3	4	5

Finally,

the sorted list is	1	2	3	4	5

Examp1e 2:

Example

80 32 31 **110** 50 **40** {← given initially}

Initially, MAX ← 80 MAX-POS ←1

After one iteration, finally; MAX ← 110, MAX-POS = 4

Numbers 110 and 40 are exchanged to get

80 32 31 **40** 50 **110**

New List, after one iteration, to be sorted is given by:

80 32 31 40 50

Initially MAX ← 80, MAX-POS ←1

Finally also MAX ← 80, MAX-POS ←1

∴ entries 80 and 50 are exchanged to get

50 32 31 40 80

New List, after second iteration, to be sorted is given by:

50 32 31 40

Initially Max ← 50, MAX-POS ←1

Finally, also Max ← 50, MAX-POS ←1

Timing Complexity for Selection Sort

- Selection Sort requires two nested for loops to complete itself, one for loop is in the function selection Sort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.
- Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

- Worst Case Time Complexity [Big-O]: $O(n^2)$
- Best Case Time Complexity [Big-omega]: $O(n^2)$
- Average Time Complexity [Big-theta]: $O(n^2)$

Advantage of Selection Sort

- Selection sort uses minimum number of swap operations O(n) among all the sorting algorithms.

Disadvantage of Selection Sort

- Selection sort is not a very efficient algorithm when data sets are large.
- This is indicated by the average and worst case complexities.

2.8 A WORST CASE OPTIMAL ALGORITHM - IMPLEMENTATION OF SELECT2

solution for the selection problem. In this problem, we are given n elements $a[1 : n]$ and are required to determine the kth-smallest element. If the partitioning element v is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the kth-smallest element is in $a[1 : j-1]$; if $k = j$, then $a[j]$ is the kth-smallest element; and if $k > j$, then the kth-smallest element is the $(k-j)$th-smallest element in $a[j+1 : n]$. The resulting algorithm is function Select1 This function places the kth-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \leq a[k]$, $1 \leq i < k$, and $a[i] \geq a[k]$, $k < i \leq n$.

Before attempting to write a pseudocode algorithm implementing Select2, we need to decide how the median of a set of size r is to be found and where we are going to store the $\lfloor n/r \rfloor$ medians of lines 8 and 9. Since, we expect to be using a small r (say $r = 5$ or 9), an efficient way to find the median of r elements is to sort them using InsertionSort(a, i, j). This algorithm is a modification of Algorithm to sort $a[i : j]$. The median is now the middle element in $a[i : j]$. A convenient place to store these medians is at the front of the array. Thus, if we are finding the kth-smallest element in $a[low : up]$, then the elements can be rearranged so that the medians are $a[low]$, $a[low+1]$, $a[low+2]$, and so on. This makes it easy to implement line 10 as a selection on consecutive elements of $a[\,]$. Function Select2 (Algorithm results from the above discussion and the replacement of the recursive calls of lines 15 and 16 by equivalent code to restart the algorithm.

```
1   Algorithm Select2(a, k, low, up)
2   // Return i such that a[i] is the kth-smallest element in
3   // a[low : up]; r is a global variable as described in the text.
4   {
5       repeat
6       {
7           n := up − low + 1; // Number of elements
8           if (n ≤ r) then
9           {
10              InsertionSort(a, low, up);
11              return low + k − 1;
12          }
13          for i := 1 to ⌊n/r⌋ do
14          {
15              InsertionSort(a, low + (i − 1) ∗ r, low + i ∗ r − 1);
16              // Collect medians in the front part of a[low : up].
17              Interchange(a, low + i − 1,
18                  low + (i − 1) ∗ r + ⌈r/2⌉ − 1);
19          }
20          j := Select2(a, ⌈⌊n/r⌋/2⌉, low, low + ⌊n/r⌋ − 1); // mm
21          Interchange(a, low, j);
22          j := Partition(a, low, up + 1);
23          if (k = (j − low + 1)) then return j;
24          else if (k < (j − low + 1)) then up := j − 1;
25              else
26              {
27                  k := k − (j − low + 1); low := j + 1;
28              }
29      } until (false);
30  }
```

2.7 STRASSEN'S MATRIX MULTIPLICATION

The Strassen's matrix multiplication algorithm finds the product C of two 2 by 2 matrices A and B with just seven multiplication as opposed to eight required by the brute force algorithm.

It is accomplished by using the following formula.

$$\begin{vmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{vmatrix} = \begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} * \begin{vmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{vmatrix}$$

$$= \begin{vmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{vmatrix}$$

Where,

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * b_{00}$$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Thus, to multiply two 2 by 2 matrices, strassen's algorithm makes seven multiplication and 18 additions/subtractions where as the brute force algorithm requires eight multiplication and four additions. These numbers should not lead us to multiplying 2 by 2 matrices by strassen's algorithm. Its importance stems from its asymptotic superiority as matrix order n goes to infinity.

Let A and B be two *n*-by-*n* matrices where n is a power of two. If *n* is not a power of two, matrices can be added with rows and column of zeros. The matrix A,B and their product is divided into 4,*n*/2 by *n*/2 submatrices each as follows

$$\begin{vmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{vmatrix} = \begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix} * \begin{vmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{vmatrix}$$

The value C_{00} can be computed as either $a_{00} * b_{00}$ or $a_{01} * b_{10}$ or as $M_1 + M_4 - M_5 + M_7$, where M_1, M_4, M_5 and M_7 are found by strassen's formula with the numbers replaced by the corresponding submatrices.

The seven products of *n*/2 and *n*/2 matrices are computed recursively by the strassen's matrix multiplication algorithm.

Algorithm Stassen's Matrix Multiplications

Algorithm

1. If n = 1 Output A × B
2. Else
3. Compute A00, B01, . . ., A11, B11 % by computing m = n/2
4. m1 ← Strassen(A00, B01 – B11)
5. m2 ← Strassen(A00+ A01, B11)
6. m3 ← Strassen(A10 + A11, B00)
7. m4 ← Strassen(A11, B10 – B00)
8. m5 ← Strassen(A00 + A11, B00 + B11)
9. m6 ← Strassen(A01 – A11, B10 + B11)
10. m7 ← Strassen(A00 – A10, B00 + B01)
11. C 00 ← m5 + m4 – m2 + m6

12. C 01 ← m1 + m2
13. C 10 ← m3 + m4
14. C 11 ← m1 + m5 – m3 – m7
15. Output C
16. End If

Example of Stassen's Matrix Multiplications

Example 1:

Example

Apply strassen's method to mulitiply the following two matrics:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 5 & 7 & 1 \\ 2 & 7 & 0 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 5 & 6 & 7 & 8 \\ 1 & 0 & 3 & 4 \\ 6 & 2 & 7 & 0 \\ 8 & 1 & 6 & 5 \end{bmatrix}$$

Solution:

The matrix is of order 4×4. Hence we will subdivide it in 2×2 submatrices. Hence we will compute $S_1, S_2, S_3, S_4, S_5, S_6, S_7$.

A_{11}	A_{12}
1 2	3 4
5 2	7 1
A_{21}	A_{22}
2 7	0 5
4 3	2 1

x

B_{11}	B_{12}
5 6	7 8
1 0	3 4
B_{21}	B_{22}
6 2	7 0
8 1	6 5

$$S_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$= \left(\begin{bmatrix} 1 & 2 \\ 5 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 5 \\ 2 & 1 \end{bmatrix} \right) \times \left(\begin{bmatrix} 5 & 6 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 7 & 0 \\ 6 & 5 \end{bmatrix} \right)$$

$$\begin{bmatrix} 1 & 7 \\ 7 & 3 \end{bmatrix} \times \begin{bmatrix} 12 & 6 \\ 7 & 5 \end{bmatrix} = \begin{bmatrix} 12 + 49 & 6 + 35 \\ 84 + 21 & 42 + 15 \end{bmatrix}$$

$$P = \begin{bmatrix} 61 & 41 \\ 105 & 57 \end{bmatrix}$$

$$S_2 = (A_{21} + A_{22}) \times B_{11}$$

$$= \left(\begin{pmatrix} 2 & 7 \\ 4 & 3 \end{pmatrix} + \begin{pmatrix} 0 & 5 \\ 2 & 1 \end{pmatrix}\right) \times \left(\begin{pmatrix} 5 & 6 \\ 1 & 0 \end{pmatrix}\right)$$

$$= \begin{pmatrix} 2 & 12 \\ 6 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 10 + 12 & 12 + 0 \\ 30 + 4 & 36 + 0 \end{pmatrix}$$

$$= \begin{pmatrix} 22 & 12 \\ 34 & 36 \end{pmatrix}$$

$S_3 = A_{11} \times (B_{12} - B_{22})$

$$= \left(\begin{pmatrix} 1 & 2 \\ 5 & 2 \end{pmatrix}\right) \times \left(\begin{pmatrix} 7 & 8 \\ 3 & 4 \end{pmatrix} - \begin{pmatrix} 7 & 0 \\ 6 & 5 \end{pmatrix}\right)$$

$$\begin{pmatrix} 1 & 2 \\ 5 & 2 \end{pmatrix} \times \begin{pmatrix} 0 & 8 \\ -3 & -1 \end{pmatrix} = \begin{pmatrix} 0 - 6 & 8 - 2 \\ 0 - 6 & 40 - 2 \end{pmatrix}$$

$$= \begin{pmatrix} -6 & 6 \\ -6 & 38 \end{pmatrix}$$

$S_4 = A_{22} \times (B_{21} - B_{11})$

$$= \left(\begin{bmatrix} 0 & 5 \\ 2 & 1 \end{bmatrix}\right) \times \left(\begin{bmatrix} 6 & 2 \\ 8 & 1 \end{bmatrix} - \begin{bmatrix} 5 & 6 \\ 1 & 0 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0 & 5 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & -4 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 0 - 35 & 0 - 5 \\ 2 - 7 & -8 - 1 \end{bmatrix}$$

$$= \begin{bmatrix} 35 & 5 \\ 9 & -7 \end{bmatrix}$$

$S_5 = (A_{11} + A_{12}) \times B_{22}$

$$= \left(\begin{bmatrix} 1 & 2 \\ 5 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 1 \end{bmatrix}\right) \times \left(\begin{bmatrix} 7 & 0 \\ 6 & 5 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 4 & 6 \\ 12 & 3 \end{bmatrix} \times \begin{bmatrix} 7 & 0 \\ 6 & 5 \end{bmatrix} = \begin{bmatrix} 28 + 36 & 0 + 30 \\ 84 + 18 & 0 + 15 \end{bmatrix}$$

$$= \begin{bmatrix} 64 & 30 \\ 102 & 15 \end{bmatrix}$$

$S_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$

$$= \left(\begin{bmatrix} 2 & 7 \\ 4 & 3 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 5 & 2 \end{bmatrix}\right) \times \left(\begin{bmatrix} 5 & 6 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 7 & 8 \\ 3 & 4 \end{bmatrix}\right)$$

$$= \begin{pmatrix} 1 & 5 \\ -1 & 1 \end{pmatrix} x \begin{pmatrix} 12 & 14 \\ 4 & 4 \end{pmatrix} = \begin{pmatrix} 12 + 20 & 14 + 20 \\ -12 + 4 & -14 + 4 \end{pmatrix}$$

$$= \begin{pmatrix} 32 & 34 \\ -8 & -10 \end{pmatrix}$$

$S_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

$$= \left(\begin{pmatrix} 3 & 4 \\ 7 & 1 \end{pmatrix} - \begin{pmatrix} 0 & 5 \\ 2 & 1 \end{pmatrix} \right) x \left(\begin{pmatrix} 6 & 2 \\ 8 & 1 \end{pmatrix} + \begin{pmatrix} 7 & 0 \\ 6 & 5 \end{pmatrix} \right)$$

$$= \begin{pmatrix} 3 & -1 \\ 5 & 0 \end{pmatrix} x \begin{pmatrix} 13 & 2 \\ 14 & 6 \end{pmatrix} = \begin{pmatrix} 39 - 14 & 6 - 6 \\ 65 - 0 & 10 - 0 \end{pmatrix}$$

$$= \begin{pmatrix} 25 & 0 \\ 65 & 10 \end{pmatrix}$$

$C_{11} = S_1 + S_4 - S_5 + S_7$

$$= \begin{pmatrix} 61 & 41 \\ 105 & 57 \end{pmatrix} + \begin{pmatrix} 35 & 5 \\ 9 & -7 \end{pmatrix} - \begin{pmatrix} 64 & 30 \\ 102 & 15 \end{pmatrix} + \begin{pmatrix} 25 & 0 \\ 65 & 10 \end{pmatrix}$$

$$= \begin{pmatrix} 96 & 46 \\ 114 & 50 \end{pmatrix} - \begin{pmatrix} 64 & 30 \\ 102 & 15 \end{pmatrix} + \begin{pmatrix} 25 & 0 \\ 65 & 10 \end{pmatrix}$$

$$= \begin{pmatrix} 57 & 16 \\ 77 & 45 \end{pmatrix}$$

$C_{12} = S_3 + S_5$

$$= \begin{pmatrix} -6 & 6 \\ -6 & 57 \end{pmatrix} + \begin{pmatrix} 64 & 30 \\ 102 & 15 \end{pmatrix}$$

$$= \begin{pmatrix} 58 & 36 \\ 96 & 53 \end{pmatrix}$$

$C_{21} = S_2 + S_4$

$$= \begin{pmatrix} 22 & 12 \\ 34 & 36 \end{pmatrix} + \begin{pmatrix} 35 & 5 \\ 9 & -7 \end{pmatrix}$$

$$= \begin{pmatrix} 77 & 17 \\ 43 & 29 \end{pmatrix}$$

$C_{22} = S_1 + S_3 - S_2 + S_6$

$$= \begin{pmatrix} 61 & 41 \\ 105 & 57 \end{pmatrix} + \begin{pmatrix} -6 & 6 \\ -6 & 38 \end{pmatrix} - \begin{pmatrix} 22 & 12 \\ 34 & 36 \end{pmatrix} + \begin{pmatrix} 32 & 34 \\ -8 & -10 \end{pmatrix}$$

$$= \begin{pmatrix} 65 & 69 \\ 57 & 49 \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$= \begin{pmatrix} \begin{bmatrix} 57 & 16 \\ 77 & 45 \end{bmatrix} \begin{bmatrix} 58 & 36 \\ 96 & 53 \end{bmatrix} \\ \begin{bmatrix} 77 & 17 \\ 43 & 29 \end{bmatrix} \begin{bmatrix} 65 & 69 \\ 57 & 49 \end{bmatrix} \end{pmatrix} = \begin{pmatrix} 57 & 16 & 58 & 36 \\ 77 & 45 & 96 & 53 \\ 77 & 17 & 65 & 69 \\ 43 & 29 & 57 & 49 \end{pmatrix}$$

$n \leq ?$

Example 2:

Example

$$\begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} x \begin{bmatrix} 2 & 7 \\ 8 & 3 \end{bmatrix}$$

a00=3, a01= 5, a10=4, a11=6, b00=2, b01=7, b10=8, b11=3 m1 = (a00 + a11) x (b00 + b11)

=(3+6) x (2+3) =9 x 5 = 45

m2 = (a10 + a11) x b00

=(4+6) x 2

=10 x 2 =20

m3 = a00 x (b01 – b11)

=3 x (7-3) = 3 x 4 =12

m4 = a11 x (b10 – b00)

=6 x (8-2) = 6 x 6 =36

m5 = (a00 + a01) x b11

= (3+5) x 3 =24

m6 = (a10 – a00) x (b00 + b01)

=(4-3) x (2+7) =9

m7 = (a01 – a11) x (b10 + b11)

= (5-6) x (8+3)

=(-1) x 11 = -11

m1+m4 - m5+m7 = 45+36-24+(-11) =81-35 =46

m3+m5=12+24 =36 m2+m4=20+36=56

m1+m3-m2+m6=45+12-20+9 =66-20 =46

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} m1 + m4 - m5 + m7 & m3 + m5 \\ m2 + m4 & m1 + m3 - m2 + m6 \end{bmatrix}$$

$$C = \begin{bmatrix} 46 & 36 \\ 56 & 46 \end{bmatrix}$$

Efficiency of Strssen's matrix multiplication

2.7.1 Efficiency of strassen's matrix multiplication

If M(n) is the number of multiplication made by strassen's algorithm in multiplying two n by n matrices where n is a power of 2, then the recurrence relation is

$M(n) = 7u(n/2)$ for $n>1$,

$M(1) = 1$

Since, $n=2^k$ yields

$M(2^k)=7\ M(2^{k-1})$

$=7[7\ M(2^{k-2})]$

$=7^2\ M(2^{k-2})$

$=7^i\ M(2^{k-i})$

$=7^k\ M(2^{k-k})$

$=7^k$

Since, $k=\log_2 n$

$M(n) = 7\ \log_2^n$

$=n \log_2 7 \approx n^{2.807}$

which is smaller than n^3 required by the brute force algorithm.

Numbers of multiplications are reduced by making extra additions.

To multiply two matrices of order n>1 , the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions of matrices of size $n/2$. When $n = 1$, no additions are made since two numbers are simply multiplied. The number of additions $A(n)$ made by the Strassen's algorithm given by recurrence

$A(n) = 7A(n/2) + 18(n/2)^2$, for $n > 1$

$A(1) = 0$

According to the Master Theorem.

$A(n) \in \Theta(n \log_2^7)$

In other words, the number of additions has the same order of growth as the number of multiplications.

As a result, Strassen's algorithm in $\Theta(n \log_2^n)$, which is a better efficiency class than $\Theta(n^3)$ or the brute force method.

Limitations of Strassen's Algorithm :-

From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication, for the following four reasons :

1. The constant factor hidden in the running time of strassen's algorithm is larger than the constant factor in the value $\Theta(n^3)$ method.

2. When the matrices are spanse, methods tailored for spause matrices are faster.

3. Strassen's algorithm is not quite as nummerically stable as the native method.

4. The submatrices formed at the levels of recursion consume space.

DESIGN AND ANALYSIS OF ALGORITHMS
UNIT III

The General Method – Container Loading – Knapsack Problem – Tree Vertex Splitting – Job Sequencing with Deadlines – Minimum Cost Spanning Trees – Prim's Algorithm – Kruskal's Algorithm – An optimal Randomized Algorithm – Optimal Storage on Tapes – Optimal Merge Pattern – Single Source Shortest Paths.

3.1. GREEDY ALGORITHM GENERAL METHOD

- The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique.
- The main function of this approach is that the decision is taken on the basis of the currently available information.
- Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.
- This technique is basically used to determine the **feasible solution** that may or may not be **optimal**. The **feasible solution** is a subset that satisfies the given criteria.
- **The optimal solution** is the solution which is the best and the most favorable solution in the subset.
- In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Characteristics of Greedy method:

- The following are the characteristics of a greedy method:
  - To construct the solution in an optimal way, this algorithm creates two sets where one set contains allthe chosen items, and another set contains the rejected items.
  - A Greedy algorithm makes good local choices in the hope that the solution should be either feasibleor optimal.
  -

Components of Greedy Algorithm:

- **The components that can be used in the greedy algorithm are:**

  - **Candidate set:** A solution that is created from the set is known as a candidate set.
  - **Selection function:** This function is used to choose the candidate or subset which can be added in thesolution.
  - **Feasibility function:** A function that is used to determine whether the candidate or subset can beused to contribute to the solution or not.
  - **Objective function:** A function is used to assign the value to the solution or the partial solution.
  - **Solution function:** This function is used to intimate whether the complete function has been reachedor not.

Applications of Greedy Algorithm:

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

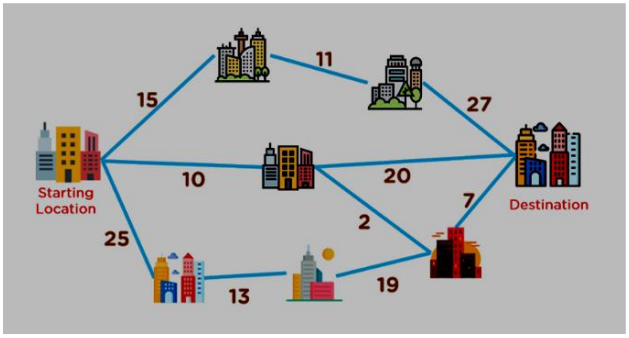
```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Types and Example:

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Knapsack Problem
- Job Scheduling Problem

Example of Greedy Algorithm

- Problem Statement: Find the best route to reach the destination city from the given starting point using a greedy method.



Output using Greedy Method:

1. Starting Location Connecting 3 cities it contains 15, 10, 25 and Choosing Min Value 10
2. After reached the vertex 10 it contains the weighted values of 20 and 2. 2 is the Min. value, choose 2 and move to.
3. Finally choose 7 and reached to the destination. So minimum cost of value to travel from source to destination is 10+ 2 + 7 = 19

3.2. CONTAINER LOADING PROBLEM

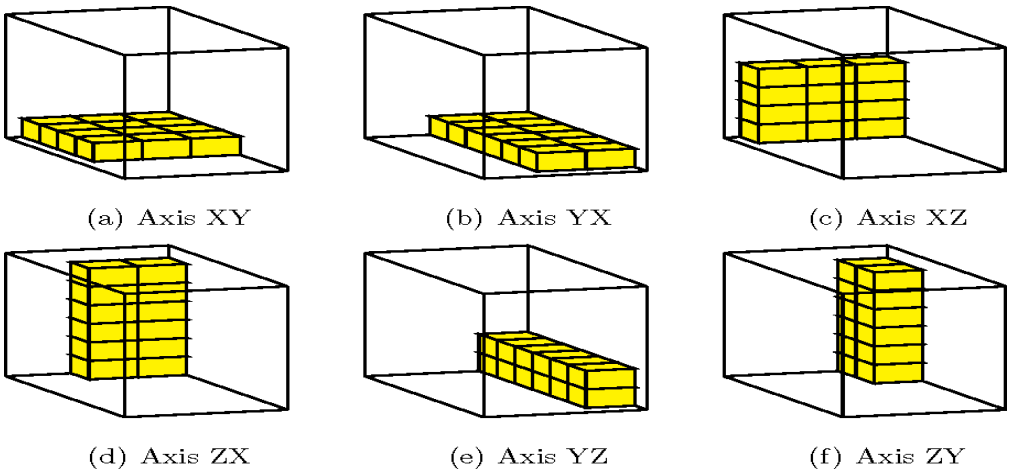
Container Loading Problem:

- The basic Container Loading Problem can be defined as the problem of placing a set of boxes into the container respecting the geometric constraints: the boxes cannot overlap and cannot exceed the dimensions of the container.
- The greedy algorithm constructs the loading plan of a single container layer by layer from the bottom up. At the initial stage, the list of available surfaces contains only the initial surface of size L x W with its initial position at height 0.

Types of Container Units and Designs for Shipping Cargo:

- ✓ Dry storage container.
- ✓ Flat rack container.
- ✓ Open top container.
- ✓ Tunnel container.
- ✓ Open side storage container.
- ✓ Double doors container.
- ✓ Refrigerated ISO containers.
- ✓ Insulated or thermal containers.

Diagrammatic Representation of Container Loading:



Container loading problem is the problem of loading a subset of rectangular boxes into a rectangular container of fixed dimensions such that the volume of the packed boxes is maximized.

Problem:

A largest ship is to be loaded with cargo is containerized and all containers are same size.
Here

Capacity of the container is C and max. No. of containers loaded in the cargo.

Check the constraint and formula is = $\sum_{i=0}^{n} wi, xi < C$
x_i = 1 = it means container loaded in the cargo
x_i = 0 = it means container not loaded in the cargo
W_i = Container Weighted

Now check the condition

$$\sum_{i=0}^{n} wi, xi \leq C$$

Take the problem

Cargo contains 8 container i.e = {W1, W2, W3, W4, W5, W6, W7, W8}

Containers contains weighted = {100, 200,50, 90, 150, 50,20, 80}

So W1=100, W2 = 200, W3=50, W4=90, W5=150, W6=50, W7=20 & W8=80

The total capacity of cargo is **400 i.e, C=400**

Stage 1: Initially is the weighted in ascending order.

{Container = Weighted}

{**7**, 3, 6, 8, 4, 1, 5, 2} = {**20**, 50, 50, 80, 90, 100, 150, 200}

Now you can take the solution:

Solution set = {0, 0, 0, 0, 0, 0, 0, 0}

Apply into formula

$wi, xi \leq C$ -> check the constraint

$20 * 1 \leq 400$ now condition is satisfied

Set the solution = {0, 0, 0, 0, 0, 0, 1, 0}

Stage 2: {7, **3**, 6, 8, 4, 1, 5, 2} = {20, **50**, 50, 80, 90, 100, 150, 200}

20 + 50 ≤ 400

70 ≤ 400 condition is satisfied

Set the solution = {0, 0, 1, 0, 0, 0, 1, 0}

Stage 3: {7, 3, **6**, 8, 4, 1, 5, 2} = {20, 50, **50**, 80, 90, 100, 150, 200}

70+ 50 ≤ 400

120 ≤ 400 condition is satisfied

Set the solution = {0, 0, 1, 0, 0, 1, 1, 0}

Stage 4: {7, 3, 6, **8**, 4, 1, 5, 2} = {20, 50, 50, **80**, 90, 100, 150, 200}

120+ 80 ≤ 400

200 ≤ 400 condition is satisfied

Set the solution = {0, 0, 1, 0, 0, 1, 1, 1}

Stage 5: {7, 3, 6, 8, **4**, 1, 5, 2} = {20, 50, 50, 80, **90**, 100, 150, 200}

200+ 90 ≤ 400

290 ≤ 400 condition is satisfied

Set the solution = {0, 0, 1, 1, 0, 1, 1, 1}

Stage 6: {7, 3, 6, 8, 4, **1**, 5, 2} = {20, 50, 50, 80, 90, **100**, 150, 200}

290+ 100 ≤ 400

390 ≤ 400 condition is satisfied

Set the solution = {1, 0, 1, 1, 0, 1, 1, 1}

Stage 7: {7, 3, 6, 8, 4, 1, **5**, 2} = {20, 50, 50, 80, 90, 100, **150**, 200}

390 + 150 ≤ 400

540 ≤ 400 condition is **Not Satisfied**

Set the solution = {1, 0, 1, 1, 0, 1, 1, 1}

Therefore container 5 is **540** ≤ 400 the condition is not satisfied and it cannot be loaded in the cargo ship.

*****Finally the container {7, 3, 6, 8, 4 and 1} = {20, 50, 50, 80, 90, 100} loaded in the cargo ship.***

*****Container {5 and 2 } i.e Weighted ={150 and 200} Not loaded in the container cargo.***

Algorithm for Container Loading

```
void containerLoading(container* c, int capacity,
                      int numberOfContainers, int* x)
{// Greedy algorithm for container loading.
 // Set x[i] = 1 iff container i, i >= 1 is loaded.
   // sort into increasing order of weight
   heapSort(c, numberOfContainers);

   int n = numberOfContainers;

   // initialize x
   for (int i = 1; i <= n; i++)
      x[i] = 0;

   // select containers in order of weight
   for (int i = 1; i <= n && c[i].weight <= capacity; i++)
   {// enough capacity for container c[i].id
      x[c[i].id] = 1;
      capacity -= c[i].weight;  // remaining capacity
   }
}
```

3.3. KNAPSACK PROBLEM

- Fractional Knapsack problem or Knapsack Problem in Greedy method. There are n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0 \leq x_i \leq 1$

of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is **m**, we require the total weight of all chosen objects to be at most **m**.

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m$$

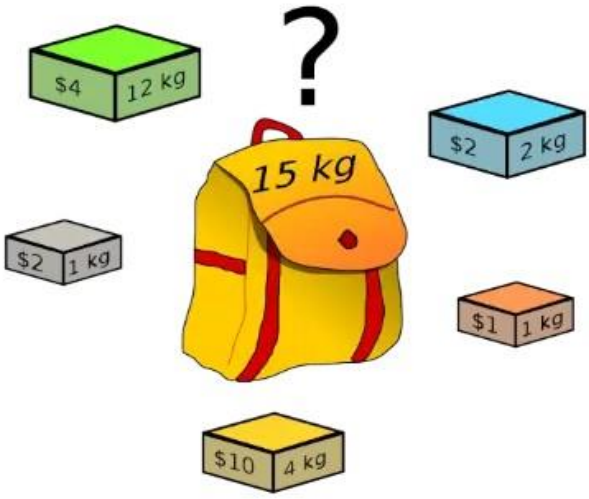
$$\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

- Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem

Fractional Knapsack Problem-

Problem-

For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

OR

Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

n = 5

w = 60 kg

(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)

(b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

OR

A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Solution-

Step-01:

Compute the value / weight ratio for each item-

Items	Weight	Value	Ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step-02:

Sort all the items in decreasing order of their value / weight ratio-

I1	I2	I5	I4	I3
(6)	(4)	(3.6)	(3.5)	(3)

Step-03:

Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	Items in Knapsack	Cost
60	Ø	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

- Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.
- Since in fractional knapsack problem, even the fraction of any item can be taken.
- So, knapsack will contain the following items-

< I1 , I2 , I5 , (20/22) I4 >

Total cost of the knapsack

= 160 + (20/27) x 77

= 160 + 70

= 230 units

Important Note-

Had the problem been a 0/1 knapsack problem, knapsack would contain the following items-

< I1 , I2 , I5 >

The knapsack's total cost would be 160 units.

A pseudo-code for solving knapsack problems using the greedy method is;

```
greedy fractional-knapsack (P[1…n], W[1…n], X[1..n]. M)
/*P[1…n] and W[1…n] contain the profit and weight of the n-objects ordered such that X[1…n]is a solution set and M is the capacity of knapsack*/
{

For j ← 1 to n doX[j]←
0
profit ← 0 // Total profit of item filled in the knapsack
weight ← 0 // Total weight of items packed in knapsacksj
← 1
While (Weight < M) // M is the knapsack capacity

{

if (weight + W[j] =< M)
X[j] = 1
weight = weight + W[j]
else{
X[j] = (M – weight)/w[j]
weight = M

}
Profit = profit + p[j] * X[j]
j++;
} // end of while
} // end of Algorithm
```

Applications

- Cutting raw materials without losing too much material
- Picking through the investments and portfolios
- Selecting assets of asset-backed securitization
- Generating keys for the Merkle-Hellman algorithm
- Cognitive Radio Networks
- Power Allocation
- Network selection for mobile nodes

Advantage:

• Will find an optimal solution if an optimal solution exists.

Disadvantage:

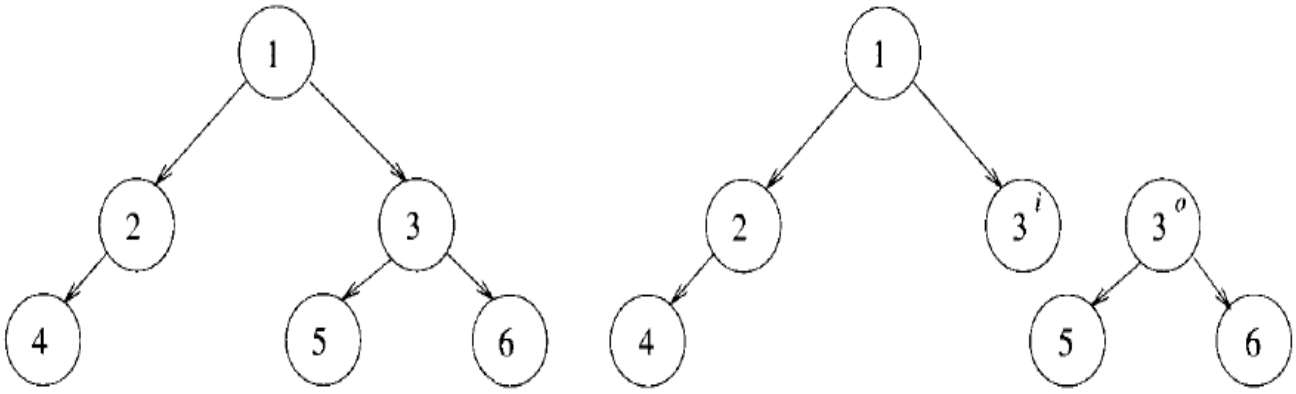
- Computationally very demanding: O (n *2 n).
- Very memory intensive.
- Number of nodes and links both grow with 2 n, node label grows with n

3.4. TREE VERTEX SPLITTING

Consider a directed binary tree each edge of which is labeled with a real number (called its *weight*). Trees with edge weights are called *weighted trees*. A weighted tree can be used, for example, to model a distribution network in which electric signals or commodities such as oil are transmitted. Nodes in the tree correspond to receiving stations and edges correspond to transmission lines. It is conceivable that in the process of transmission some loss occurs (drop in voltage in the case of electric signals or drop in pressure in the case of oil). Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network may not be able to tolerate losses beyond a certain level. In places where the loss exceeds the tolerance level, boosters have to be placed. Given a network and a loss tolerance level, the *Tree Vertex Splitting Problem (TVSP)* is to determine an optimal placement of boosters. It is assumed that the boosters can only be placed in the nodes of the tree.

The TVSP can be specified more precisely as follows: Let $T = (V, E, w)$ be a weighted directed tree, where V is the vertex set, E is the edge set, and w is the weight function for the edges. In particular, $w(i, j)$ is the weight of the edge $\langle i, j \rangle \in E$. The weight $w(i, j)$ is undefined for any $\langle i, j \rangle \notin E$. A *source vertex* is a vertex with in-degree zero, and a *sink vertex* is a vertex with out-degree zero. For any path P in the tree, its *delay*, $d(P)$, is defined to be the sum of the weights on that path. The delay of the tree T, $d(T)$, is the maximum of all the path delays.

Let T/X be the forest that results when each vertex u in X is split into two nodes u^i and u^o such that all the edges $\langle u, j \rangle \in E$ ($\langle j, u \rangle \in E$) are replaced by edges of the form $\langle u^o, j \rangle$ ($\langle j, u^i \rangle$). In other words, outbound edges from u now leave from u^o and inbound edges to u now enter at u^i. Figure 4.1 shows a tree before and after splitting the node 3. A node that gets split corresponds to a booster station. The TVSP is to identify a set $X \subseteq V$ of minimum cardinality for which $d(T/X) \leq \delta$, for some specified tolerance limit δ. Note that the TVSP has a solution only if the maximum edge weight is $\leq \delta$. Also note that the TVSP naturally fits the subset paradigm.



A tree before and after splitting the node 3

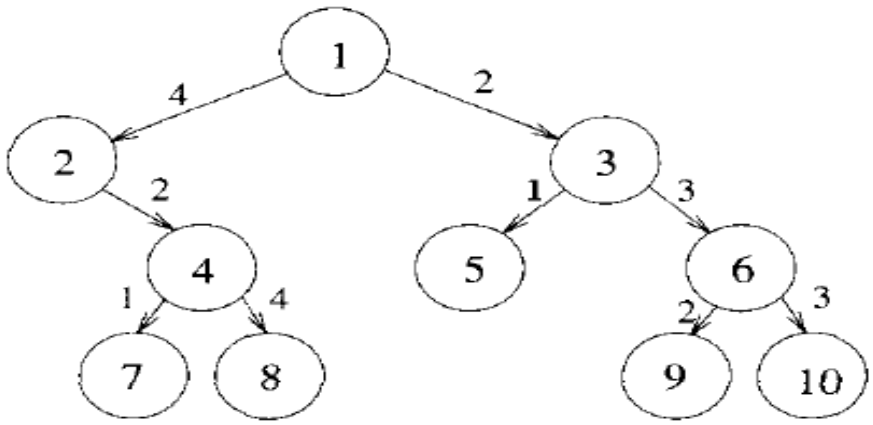
```
Algorithm TVS(T, δ)
// Determine and output the nodes to be split.
// w() is the weighting function for the edges.
{
    if (T ≠ 0) then
    {
        d[T] := 0;
        for each child v of T do
        {
            TVS(v, δ);
            d[T] := max{d[T], d[v] + w(T, v)};
        }
        if ((T is not the root) and
                (d[T] + w(parent(T), T) > δ)) then
        {
            write (T); d[T] := 0;
        }
    }
}
```

Example:

In the tree of Figure let $\delta = 5$. For each of the leaf nodes $7, 8, 5, 9$, and 10 the delay is zero. The delay for any node is computed only after the delays for its children have been determined. Let u be any node and $C(u)$ be the set of all children of u. Then $d(u)$ is given by

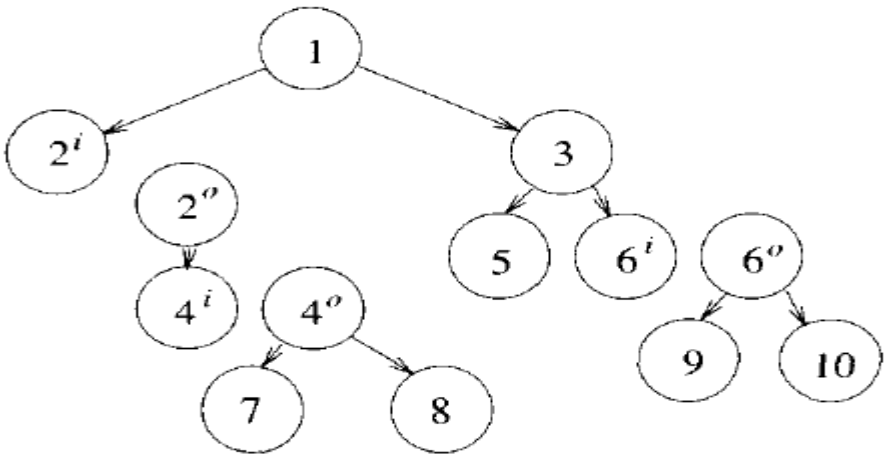
$$d(u) = \max_{v \in C(u)} \{d(v) + w(u, v)\}$$

Using the above formula, for the tree of Figure $d(4) = 4$. Since $d(4) + w(2, 4) = 6 > \delta$, node 4 gets split. We set $d(4) = 0$. Now $d(2)$ can be



computed and is equal to 2. Since $d(2) + w(1, 2)$ exceeds δ, node 2 gets split and $d(2)$ is set to zero. Then $d(6)$ is equal to 3. Also, since $d(6) + w(3, 6) > \delta$, node 6 has to be split. Set $d(6)$ to zero. Now $d(3)$ is computed as 3. Finally, $d(1)$ is computed as 5.

Figure shows the final tree that results after splitting the nodes 2, 4, and 6. This algorithm is described in Algorithm , which is invoked as TVS($root, \delta$), *root* being the root of the tree. The order in which TVS visits (i.e., computes the delay values of) the nodes of the tree is called the *post order*



The final tree after splitting the nodes 2, 4, and 6

3.5. JOB SEQUENCING WITH DEADLINES

We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadlines, and it should yield a maximum profit.

Points To remember:

- To complete a job, one has to process the job or a action for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.
- If we select a job at that time ,
- Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.
- So the waiting time and the processing time should be less than or equal to the dead line of the job.

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1) step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

Problem-

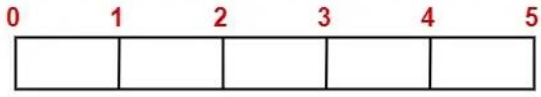
Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Step-02:

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



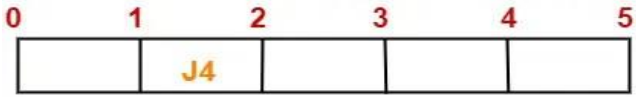
Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

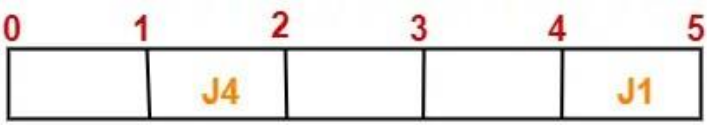
Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



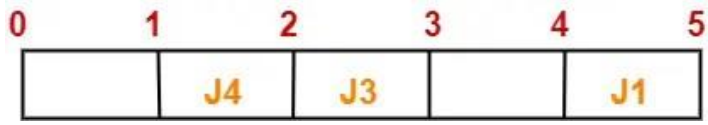
Step-04:

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



Step-06:

- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-

Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

The optimal schedule is-

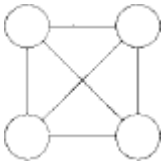
J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

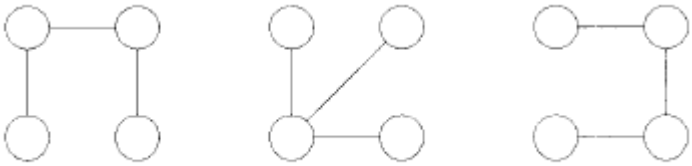
3.6 MINIMUM COST SPANNING TREES – PRIM'S ALGORITHM – KRUSKAL'S ALGORITHM

Minimum Cost Spanning Tree:

Definition Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of G is a *spanning tree* of G iff t is a tree.



Graph



Spanning Tree of the above Graph

The problem is to generate a graph G'= (V,E) where 'E' is the subset of E,G' is a

Minimum spanning tree:

- Each and every edge will contain the given non-negative length .connect all the nodes with edge present in set E? and weight has to be minimum.

NOTE:

- We have to visit all the nodes.
- The subset tree (i.e) any connected graph with 'N' vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

- A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph.
- A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

Minimum cost spanning tree:

- The cost of a spanning tree is the sum of cost of the edges in that trees.
- There are 2 method to determine a minimum cost spanning tree are

1. Kruskal's Algorithm
2. Prom's Algorithm.

Advantages:

- The minimum spanning tree (MST) is an important concept in network design and optimization.
- The main benefit of finding the MST in a network is that it provides the most cost-effective way to connect all nodes in the network while minimizing the total weight (or cost) of the edges.

Applications of Minimum Spanning Tree Problem

- Network design.
- Approximation algorithms for NP-hard problems.
- Indirect applications.
- Cluster analysis.
- Image segmentation
- To find paths in the map
- To design networks like telecommunication networks, water supply networks, and electrical grids.

3.6.1. PRIM'S ALGORITHM

- **Spanning tree** - A spanning tree is the subgraph of an undirected connected graph.
- **Minimum Spanning Tree** - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.
- **Definition: Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph.
- **Prim's algorithm** finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Algorithm

Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
Step 4: Add the selected edge and the vertex to the minimum spanning tree T
[END OF LOOP]
Step 5: EXIT

Applications:

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

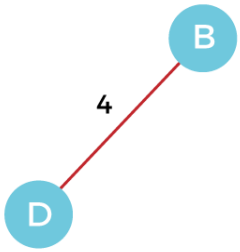
Example:

- Suppose, a weighted graph is -

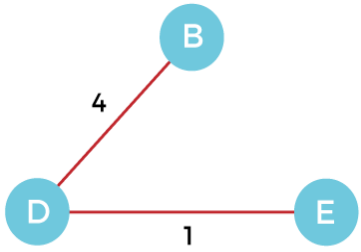
Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

Step 2 - Now, o choose and add the shortest edge from vertex B.

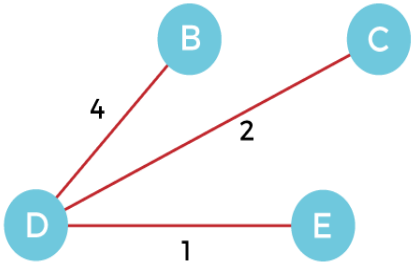
There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



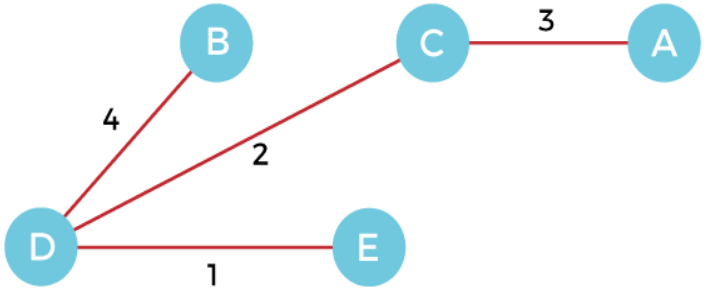
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Time Complexity:

The time complexity of the Prim's Algorithm is **O ((V + E) l o g V).**

Algorithm for Prim's Algorithm:

```
Algorithm Prim(E, cost, n, t)
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n - 1 do
    { // Find n - 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

3.6.2. KRUSKAL'S ALGORITHM

- In Kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle.
- Invented by Joseph Kruskal
- Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree.
- Kruskal's greedy algorithm finds a minimum spanning tree for a weighted, undirected graph.
- The algorithm starts with a forest consisting of the individual nodes of the graph and then finds the cheapest edge from each node and adds it to the forest.
- Sort all the edges from low weight to high.
- The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

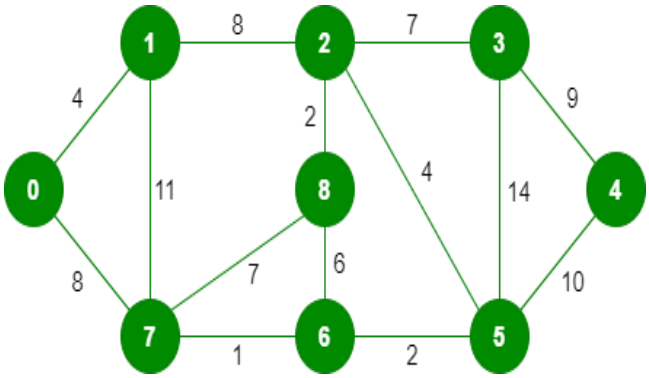
- In this algorithm, a minimum cost-spanning tree 'T' is built edge by edge.
- Edge are considered for inclusion in 'T' in increasing order of their cost.
- An edge is included in 'T' if it doesn't form a cycle with edge already in T.
- To find the minimum cost spanning tree the edge are inserted to tree in increasing order of their cost.

Advantages:

- To find the subset of edges that generate the tree and includes each and every vertex where the sum of all weight of the edges is a minimum.
  Kruskal algorithm is suitable for sparse graphs (low number of edges).

Ex: The Given Graph is below find the Minimum Cost spanning Tree using Kruskal's Method.

Illustration: The sample Input Graph as shown below



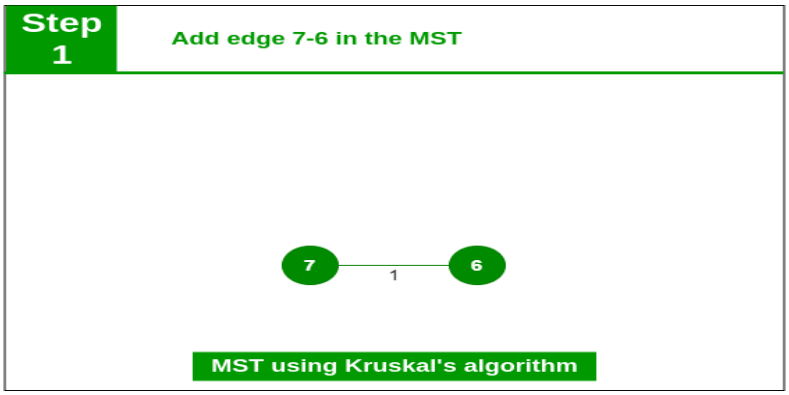
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

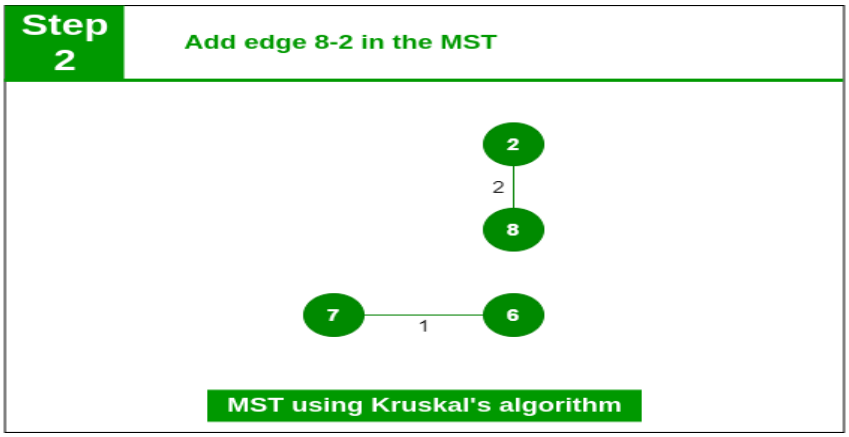
Now pick all edges one by one from the sorted list of edges

Step 1: Pick edge 7-6. No cycle is formed, include it.



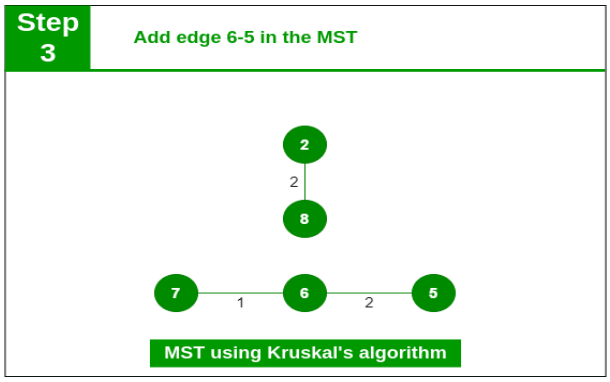
Add edge 7-6 in the MST

Step 2: Pick edge 8-2. No cycle is formed, include it.



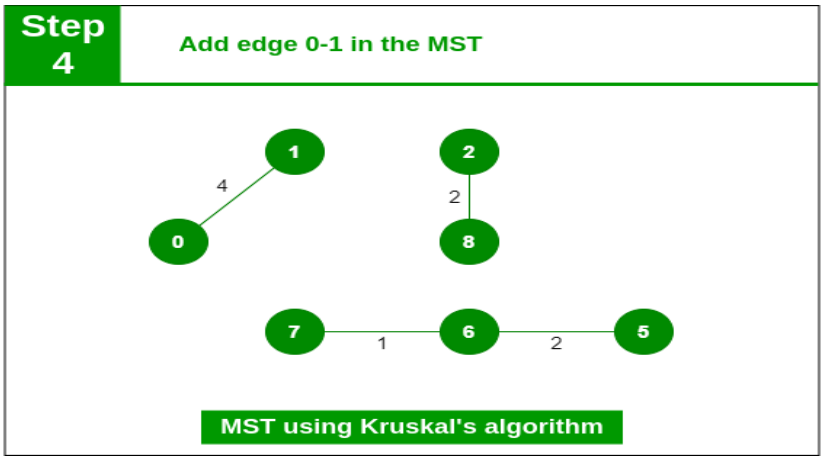
Add edge 8-2 in the MST

Step 3: Pick edge 6-5. No cycle is formed, include it.



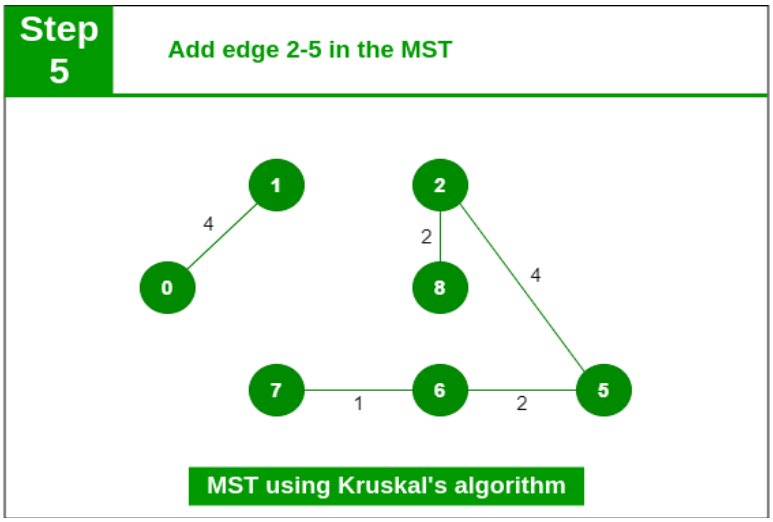
Add edge 6-5 in the MST

Step 4: Pick edge 0-1. No cycle is formed, include it.



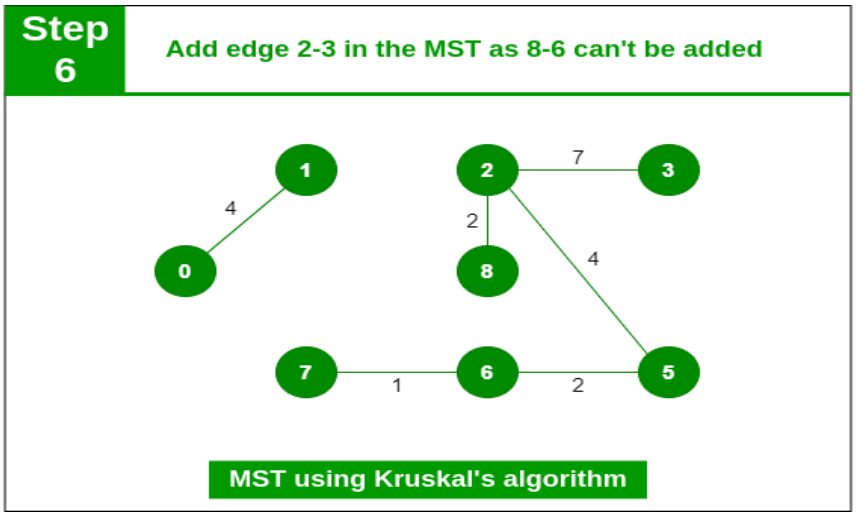
Add edge 0-1 in the MST

Step 5: Pick edge 2-5. No cycle is formed, include it.



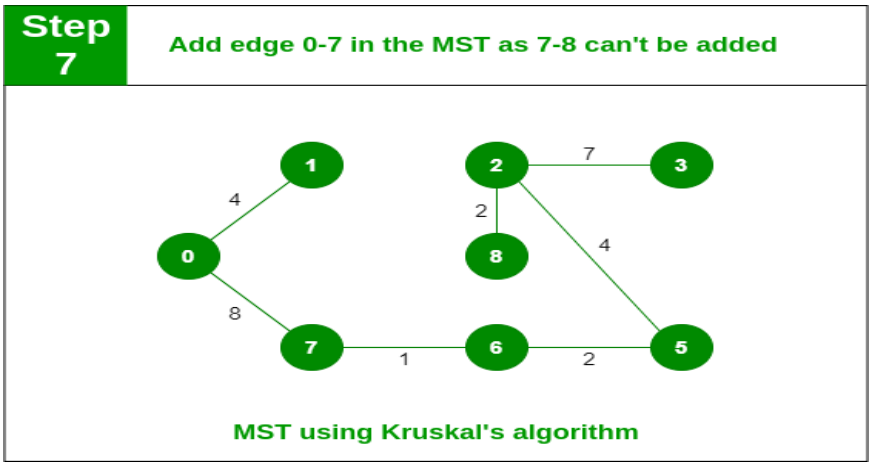
Add edge 2-5 in the MST

Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.



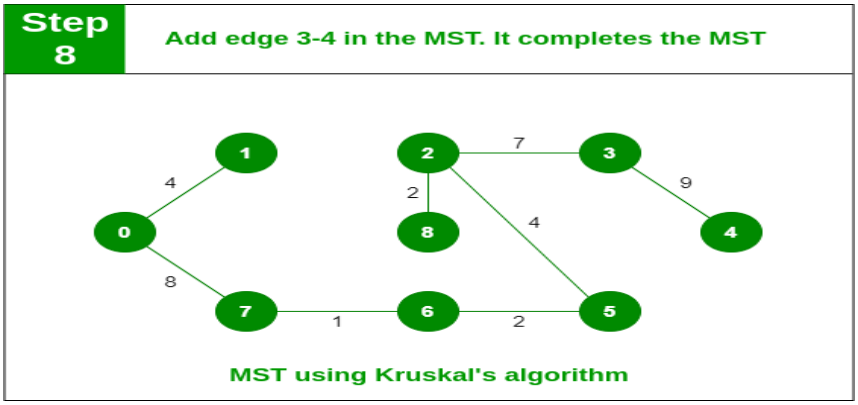
Add edge 2-3 in the MST

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



Add edge 0-7 in MST

Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.



Algorithm for Krusal's Algorithm:

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1)  and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

Time Complexity:

- O(E logE) or O(V logV) is the time complexity of the Kruskal algorithm.

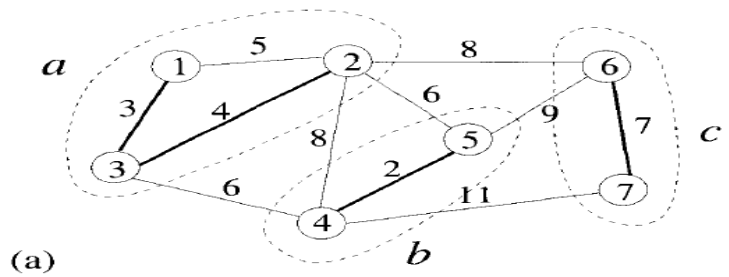
Compare Prim's and Kruskal's Algorithm:

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V2), V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components.
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.
It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.
Prim's algorithm prefer list data structures.	Kruskal's algorithm prefer heap data structures.

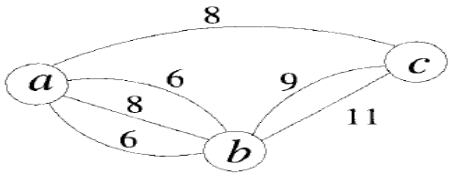
3.7. AN OPTIMAL RANDOMIZED ALGORITHM

Any algorithm for finding the minimum-cost spanning tree of a given graph $G(V, E)$ will have to spend $\Omega(|V| + |E|)$ time in the worst case, since it has to examine each node and each edge at least once before determining the correct answer. A randomized Las Vegas algorithm that runs in time $\tilde{O}(|V| + |E|)$ can be devised as follows: (1) Randomly sample m edges from G (for some suitable m). (2) Let G' be the induced subgraph; that is, G' has V as its node set and the sampled edges in its edge set. The subgraph G' need not be connected. Recursively find a minimum-cost spanning tree for each component of G'. Let F be the resultant *minimum-cost spanning forest* of G'. (3) Using F, eliminate certain edges (called the *F-heavy edges*) of G that cannot possibly be in a minimum-cost spanning tree. Let G'' be the graph that results from G after elimination of the F-heavy edges. (4) Recursively find a minimum-cost spanning tree for G''. This will also be a minimum-cost spanning tree for G.

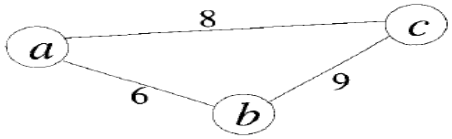
Steps 1 to 3 are useful in reducing the number of edges in G. The algorithm can be speeded up further if we can reduce the number of nodes in the input graph as well. Such a node elimination can be effected using the *Borůvka steps*. In a Borůvka step, for each node an incident edge with minimum weight is chosen. For example in **Fig 3.7 (a)** , the edge $(1, 3)$ is



(a)



(b)



(c)

chosen for node 1, the edge $(6, 7)$ is chosen for node 7, and so on. All the chosen edges are shown with thick lines. The connected components of the induced graph are found. In the example of Figure (a), the nodes 1, 2, and 3 form one component, the nodes 4 and 5 form a second component, and the nodes 6 and 7 form another component. Replace each component with a single node. The component with nodes 1, 2, and 3 is replaced with the node a. The other two components are replaced with the nodes b and c, respectively. Edges within the individual components are thrown away. The resultant graph is shown in Figure (b). In this graph keep only an edge of minimum weight between any two nodes. Delete any isolated nodes.

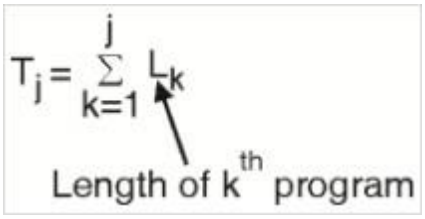
Since an edge is chosen for every node, the number of nodes after one Borůvka step reduces by a factor of at least two. A minimum-cost spanning tree for the reduced graph can be extended easily to get a minimum-cost spanning tree for the original graph. If E' is the set of edges in the minimum-cost spanning tree of the reduced graph, we simply include into E' the edges chosen in the Borůvka step to obtain the minimum-cost spanning tree edges for the original graph. In the example of Figure , a minimum-cost spanning tree for (c) will consist of the edges (a, b) and (b, c). Thus a minimum-cost spanning tree for the graph of (a) will have the edges: $(1, 3), (3, 2), (4, 5), (6, 7), (3, 4)$, and $(2, 6)$. More details of the algorithms are given below.

3.8. OPTIMAL STORAGE ON TAPES

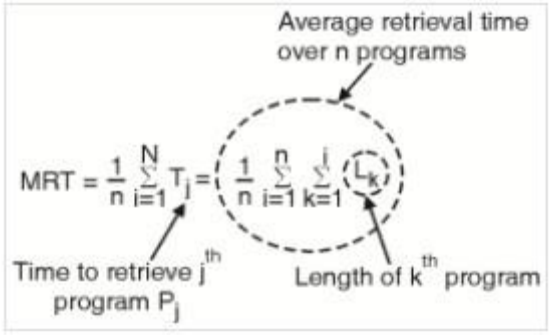
There are n programs that are to be stored on a computer tape of length l. Associated with each program i is a length $l_i, 1 \leq i \leq n$. Clearly, all programs can be stored on the tape if and only if the sum of the lengths of the programs is at most l. We assume that whenever a program is to be retrieved from this tape, the tape is initially positioned at the front. Hence, if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$, the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \leq k \leq j} l_{i_k}$. If all programs are retrieved equally often, then the expected or *mean retrieval time* (MRT) is $(1/n) \sum_{1 \leq j \leq n} t_j$. In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing $d(I) = \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq j} l_{i_k}$.

Optimal Storage on Tapes Problem: Given n programs P_1, P_2, …, P_n of length L_1, L_2, …, L_n respectively, store them on a tap of length L such that Mean Retrieval Time (MRT) is a minimum.

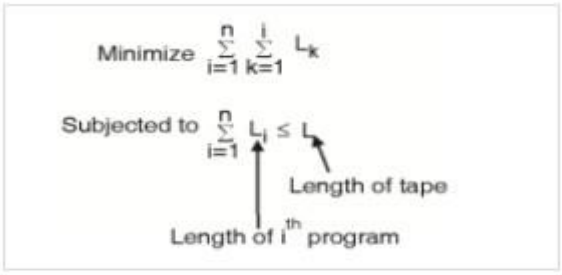
The retrieval time of the jth program is a summation of the length of first j programs on tap. Let Tj bethe time to retrieve program Pj. The retrieval time of P_j is computed as,



Mean retrieval time of n programs is the average time required to retrieve any program. It is required tostore programs in an order such that their Mean Retrieval Time is minimum. MRT is computed as,



Optimal storage on tape is minimization problem which,



- In this case, we have to find the permutation of the program order which minimizes the MRT after storing all programs on *single tape* only.
- There are many permutations of programs. Each gives a different MRT. Consider three programs **(P1, P2, P3) with a length of (L1, L2, L3) = (5, 10, 2).**
- Let's find the MRT for different permutations. 6 permutations are possible for 3 items. The Mean Retrieval Time for each permutation is listed in the following table.

Ordering	**Mean Retrieval Time** (**MRT**)

P_1, P_2, P_3	((5) + (5 + 10) + (5 + 10 + 2)) / 3 = 37 / 3
P_1, P_3, P_2	((5) + (5 + 2) + (5 + 2 + 10)) = 29 / 3
P_2, P_1, P_3	((10) + (10 + 5) + (10 + 5 + 2)) = 42 / 3
P_2, P_3, P_1	((10) + (10 + 2) + (10 + 2 + 5)) = 39 / 3
P_3, P_1, P_2	((2) + (2 + 5) + (2 + 5 + 10)) = 26 / 3
P_3, P_2, P_1	((2) + (2 + 10) + (2 + 10 + 5)) = 31 / 3

- It should be observed from the above table that the MRT is 26/3, The Optimal Storage Pattern is P3,P1,P2

```
Algorithm Store(n, m)
// n is the number of programs and m the number of tapes.
{
    j := 0; // Next tape to store on
    for i := 1 to n do
    {
        write ("append program", i,
            "to permutation for tape", j);
        j := (j + 1) mod m;
    }
}
```

3.9.OPTIMAL MERGE PATTERN

- **Optimal merge pattern** is a pattern that relates to the merging of two or more sorted files in a single
- sorted file. This type of merging can be done by the two-way merging method.
- If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.
- There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time.The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, **f (i)** represents the number of records in each file and **d (i)** represents the depth.

files x_1, x_2, x_3, and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then we could merge y_1 and x_3 to get y_2. Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 and get the desired sorted file. Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Optimal merge pattern example

Given a set of unsorted files: 5, 3, 2, 7, 9, 13

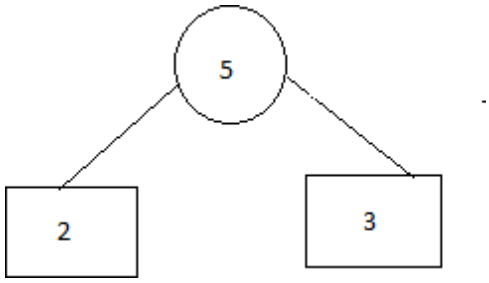
Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13

After this, pick two smallest numbers and repeat this until we left with only one number.

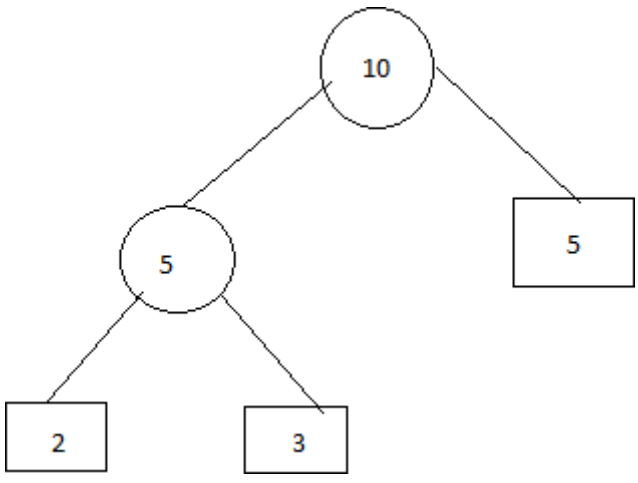
Now follow following

steps:Step 1: Insert 2, 3

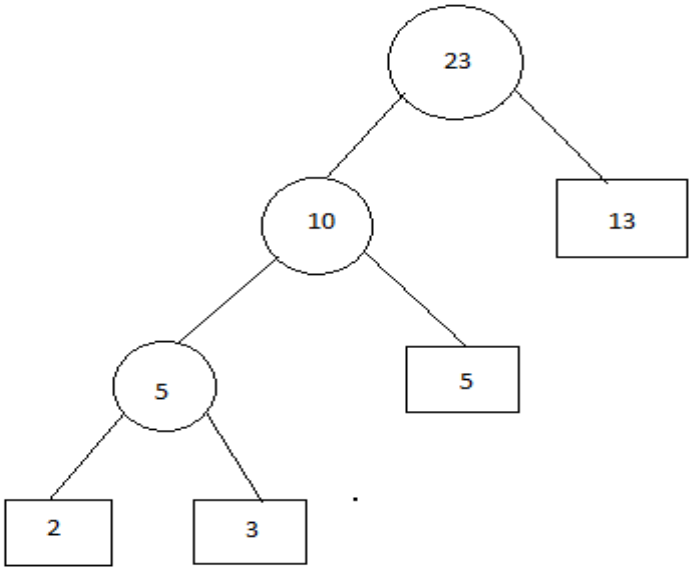
Step 2:



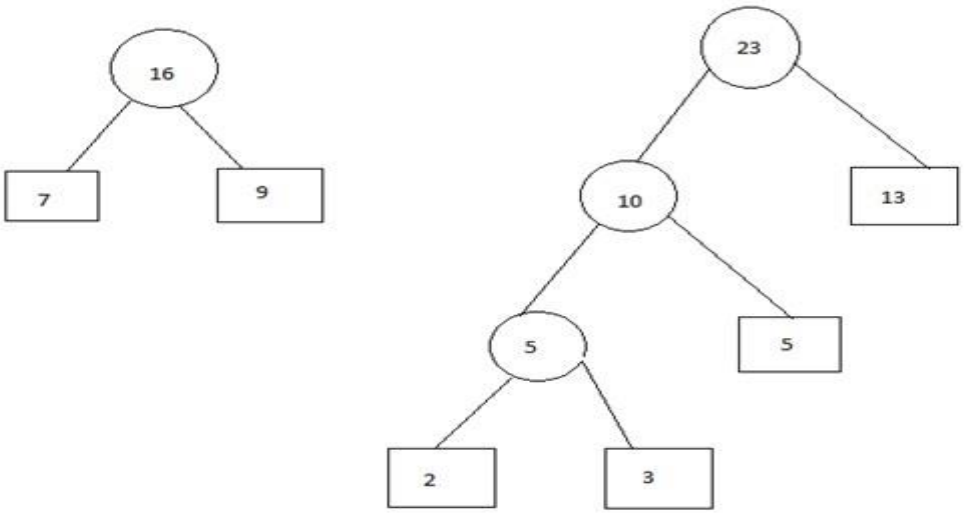
Step 3: Insert 5



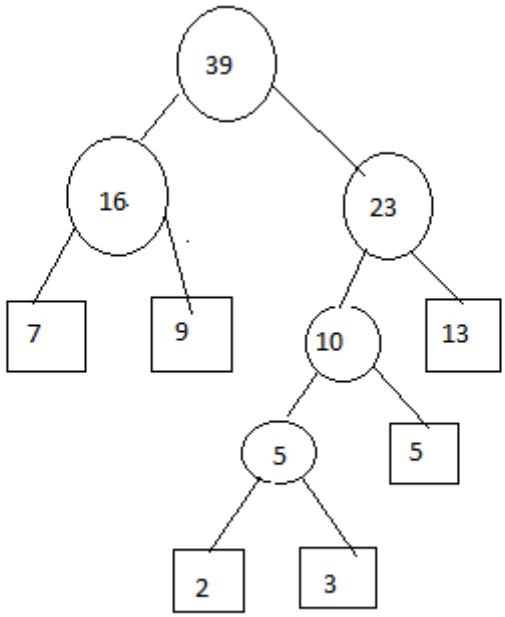
Step 4: Insert 13



Step 5: Insert 7 and 9



Step 6:



So, The merging cost = 5 + 10 + 16 + 23 + 39 = 93

Algorithm for optimal merge pattern:

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

Algorithm Tree(n)
// list is a global list of n single node
// binary trees as described above.
{
    for i := 1 to n − 1 do
    {
        pt := new treenode; // Get a new tree node.
        (pt → lchild) := Least(list); // Merge two trees with
        (pt → rchild) := Least(list); // smallest lengths.
        (pt → weight) := ((pt → lchild) → weight)
                  +((pt → rchild) → weight);
        Insert(list, pt);
    }
    return Least(list); // Tree left in list is the merge tree.
}
```

3.10. SINGLE SOURCE SHORTEST PATHS

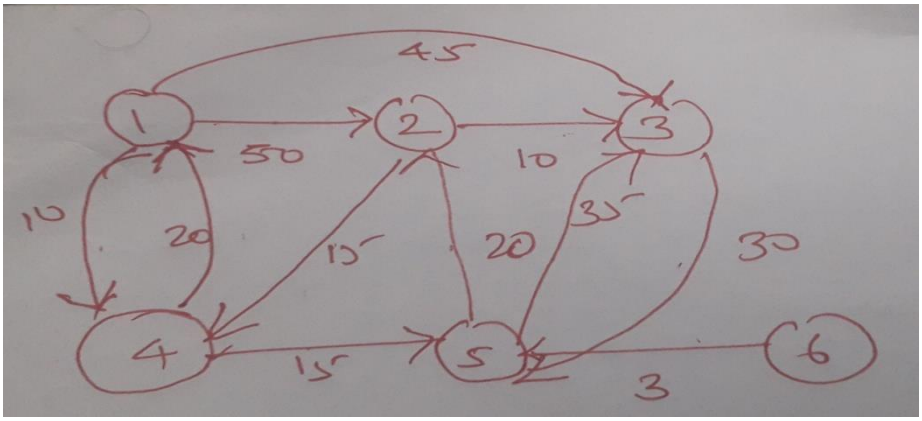
- The Single-Source Shortest Path (SSSP) problem consists of finding the shortest paths between a given vertex v and all other vertices in the graph.
- Algorithms such as Breadth-First-Search (BFS) for unweighted graphs or Dijkstra solve this problem.
- Bellman-Ford and Dijkstra's algorithms are powerful tools for finding the shortest path in a graph or network.
- Dijkstra Algorithm is a graph algorithm for finding the shortest path from a source node to all other nodes in a graph(single source shortest path).
- It is a type of greedy algorithm. It only works on weighted graphs with positive weights.
- Dijkstra's Algorithm is also known as Single Source Shortest Path (SSSP) problem.
- It is used to find the shortest path from source node to destination node in graph. The graph is widely accepted data structure to represent distance map.

Bellman-Ford Algorithm

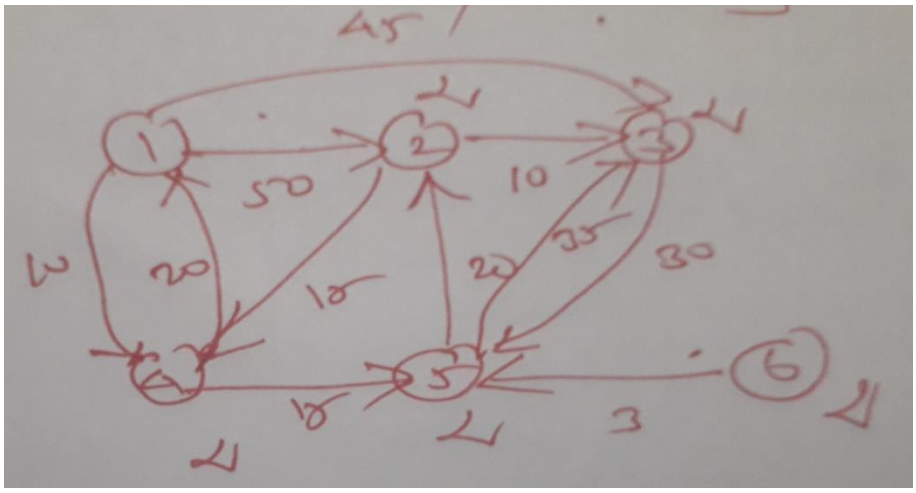
- Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs.
- A Bellman-Ford algorithm is also guaranteed to find the shortest path in a graph, similar to Dijkstra's algorithm.
- Although Bellman-Ford is slower than Dijkstra's algorithm, it is capable of handling graphs with negative edge weights, which makes it more versatile.
- The shortest path cannot be found if there exists a negative cycle in the graph.
- As a result, Bellman-Ford is also capable of detecting negative cycles, which is an important feature.

Single-source shortest path: Dijkstra's Algorithm:

- Graphs can be used to represent the highway structure of a state or country withvertices representing cities and edges representing sections of highway.
- The edges can thenbe assigned weights which may be either the distance between the two cities connected bythe edge or the average time to drive along that section of highway.
-

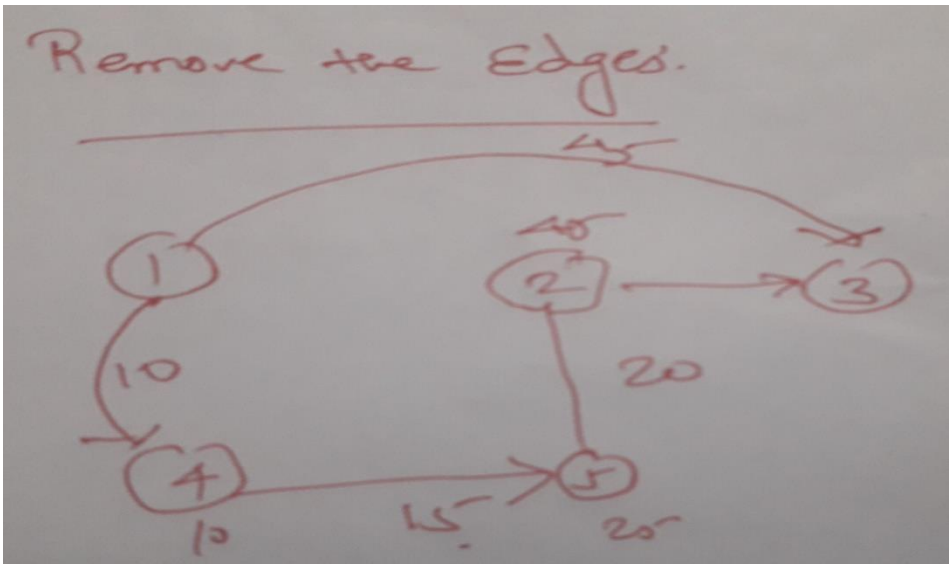


- A Direct Graph G={V,E} such that G is a Graph, V is a set of vertices and E is a set of Edges.
- V- Vertices = {V1, V2, V3, V4, V5, V6}
- E-Edges = {(1,2) (2,3) (1,3)(1,4) (4,1)(4,5)(5,2)(2,4)(3,5)(5,3)(6,5)}



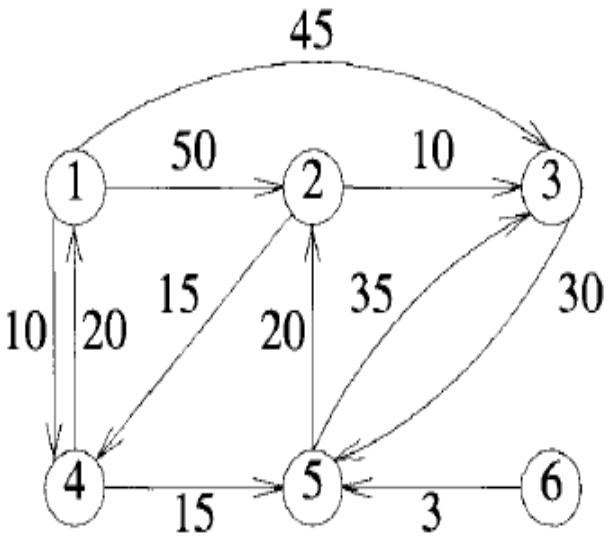
Path				Length
	Source		Destination	
1	{ 1	------	4}	10
2	{ 1	-- 4 ---	5}	25
3	{ 1	--- 4---5---	2}	45
4	{ 1	------	3}	45
5	{ 6}			∝

Result :



Algorithm for single Source Shortest Paths:

```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n − 1 do
    {
        // Determine n − 1 paths from v.
        Choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
    }
}
```



(a) Graph

	Path	*Length*
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT – IV

DYNAMIC PROGRAMMING, TRAVERSAL & SEARCHING: General method- Multi stage graphs- All pairs shortest path problem- String Editing- 0/1 knapsack problem- Reliability design - Travelling sales person problem. Techniques for Binary Trees-Techniques for Graphs-BFS-DFS.

4.0. DYNAMIC PROGRAMMING

- Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions.
- In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.
- When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations,* that enable us to solve the problem in an efficient way.
- Dynamic programming is based on the principle of optimality (also coined by Bellman).
- The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself isconstructed.

- Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal.
- Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered
- The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently.
- In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems.
- Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

4.1. DYNAMIC PROGRAMMING

- Dynamic Programming is an approach to solve problems by dividing the main complex problem into smaller parts, and then using these to build up the final solution.
- Dynamic programming is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query.
- These subproblems then overlap with one another as you find solutions by solving the same subproblem repeatedly.
- Dynamic programming can be used to solve problems with overlapping subproblems.
- There are three steps in finding a dynamic programming solution to a problem:
  - ✓ (i) Define a class of subproblems,
  - ✓ (ii) give a recurrence based on solving each subproblem in terms of simpler subproblems, and
  - ✓ (iii) Give an algorithm for computing the recurrence.

Advantages:

- ✓ Exploiting the structure and properties of the network optimization problems,
- ✓ Reducing computational complexity and memory requirements, and
- ✓ Providing exact solutions or lower bounds for the pricing problem.

Characteristics of Dynamic Programming:

Aspect	Dynamic Programming
Approach	Bottom-up (starting from base cases)
Subproblem Solving	Solves each subproblem only once
Overlapping Subproblems	Exploits overlapping substructures
Time Complexity	Generally more efficient due to memorization and avoiding duplicates

General Characteristics of Dynamic Programming:

1) The problem can be divided into stages with a policy decision required at each stage.

2) Each stage has number of states associated with it.

3) Given the current stage an optimal policy for the remaining stages is independent of the policy adopted.

4) The solution procedure begins be finding the optimal policy for each state of the last stage.

5) A recursive relation is available which identifies the optimal policy for each stage with n stages remaining

given the optimal policy for each stage with (n-1) stages remaining.

Applications of Dynamic Programming:

- ✓ Longest Common Subsequence.
- ✓ Finding Shortest Path.
- ✓ Finding Maximum Profit with other Fixed Constraints. (0/1 knapsack problem)
- ✓ Job Scheduling in Processor.
- ✓ Bioinformatics.
- ✓ Optimal search solutions.
- ✓ Matrix Chain Manipulation
- ✓ Optimal Binary Search Trees
- ✓ Multistage Graph
- ✓ Travelling Sales Man Problem

Real life example of dynamic programming

- ✓ Optimization,
- ✓ Graph theory,
- ✓ image processing,
- ✓ machine learning,
- ✓ cryptography

Example:

- ✓ Fibonacci number series
- ✓ Knapsack problem
- ✓ Tower of Hanoi
- ✓ All pair shortest path by Floyd-Warshall and Bellman Ford
- ✓ Shortest path by Dijkstra
- ✓ Project scheduling

Advantages:

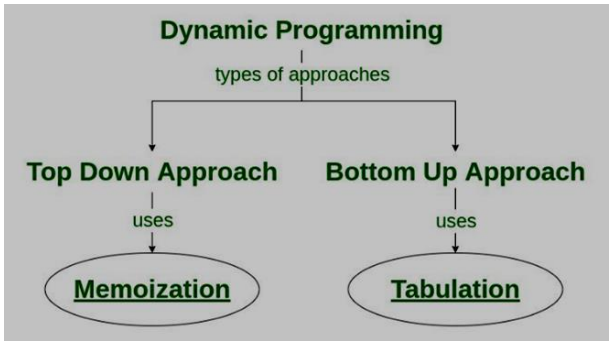
- ✓ The main use of dynamic programming is to solve optimization problems.
- ✓ To find out the minimum or the maximum solution of a problem.
- ✓ The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.
- ✓ Dynamic programming is more efficient for problems that have many overlapping subproblems, as it avoids redundant computations and saves time.

Disadvantages:

- ✓ It also requires more memory and space, as it stores all the results of the subproblems, even if some of them are not needed.
- ✓ Dynamic programming uses recursion, which requires more memory in the call stack, and leads to a stack overflow condition in the runtime.
- ✓ It takes memory to store the solutions of each sub-problem. There is no guarantee that the stored value will be used later in execution.

GENERAL METHOD

- ✓ There are two methods:
  - i) Top – Down Method
  - ii) Bottom – up Method
- ✓ **Top-down method:** The top-down method solves the overall problem before you break it down into subproblems.
- ✓ **Bottom-up method**: In the bottom-up method, or tabulation method, you solve all the related sub-problems first instead of applying recursion.



4.2. MULTI STAGE GRAPHS

- A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into k $\geq$ 2 disjoint sets V_i, 1 $\leq$ i $\leq$ k. In addition, if <u, v> is an edge in E, then u $\in V_i$ and v $\in V_{i+1}$ for some i, 1 $\leq$ i < k.
- Let the vertex 's' is the source, and 't' the sink. Let c (i, j) be the cost of edge <i, j>. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.
- A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k – 2 decisions. The i^{th}

decision involves determining which vertex in v_{i+1}, 1 $\leq$ i $\leq$ k - 2, is to be on the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\text{cost}(i, j) = \min_{\substack{l \in V_{i+1} \\ <j, l> \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

ALGORITHM:

Algorithm Fgraph (G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order or stages. E is a set of edges and c [i, j]
// is the cost of (i, j). p [1 : k] is a minimum cost path.

```
{
        cost [n] := 0.0;
        for j:= n - 1 to 1 step – 1 do
        {                                                   // compute cost [j]
                let r be a vertex such that (j, r) is an edge
                of G and c [j, r] + cost [r] is minimum; cost
                [j] := c [j, r] + cost [r];
                d [j] := r:
        }
        p [1] := 1; p [k]  := n;                            // Find a minimum cost path.
        for j := 2 to k - 1 do p [j] := d [p [j -1]];
}
```

The multistage graph problem can also be solved using the backward approach. Let bp(i, j) be a minimum cost path from vertex s to j vertex in V_i. Let Bcost(i, j) be the cost of bp(i, j). From the backward approach we obtain:

$$\text{Bcost}(i, j) = \min_{\substack{l \in V_{i-1} \\ <l, j> \in E}} \{\text{Bcost}(i - 1, l) + c(l, j)\}$$

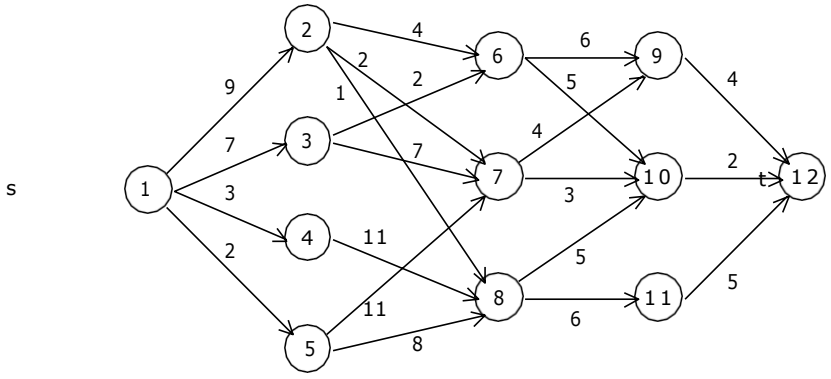
Algorithm Bgraph (G, k, n, p)
// Same function as Fgraph
{
Bcost [1] := 0.0;
for j := 2 to n do
{ // Compute Bcost [j].
Let r be such that (r, j) is an edge of G
and Bcost [r] + c [r, j] is minimum;
Bcost [j] := Bcost [r] + c [r, j];
D [j] := r;
} //find a minimum cost path
p [1] := 1; p [k] := n;
for j:= k - 1 to 2 do p [j] := d [p [j + 1]];
}

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has $|E|$ edges, then the time for the first for loop is $\Phi(|V| + |E|)$.

EXAMPLE:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.



FORWARD APPROACH:

We use the following equation to find the minimum cost path from s to t:

$$\text{cost}(i, j) = \min_{l \in V_{i+1},\ \langle j, l \rangle \in E} \{c(j, l) + \text{cost}(i + 1, l)\}$$

$$\text{cost}(1, 1) = \min \{c(1, 2) + \text{cost}(2, 2), c(1, 3) + \text{cost}(2, 3), c(1, 4) + \text{cost}(2, 4), c(1, 5) + \text{cost}(2, 5)\}$$

$$= \min \{9 + \text{cost}(2, 2), 7 + \text{cost}(2, 3), 3 + \text{cost}(2, 4), 2 + \text{cost}(2, 5)\}$$

Now first starting with,

cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3, 8)}
= min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3,8)}

---

cost (3, 6) = min {c (6, 9) + cost (4, 9), c (6, 10) + cost (4,10)}
= min {6 + cost (4, 9), 5 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0) = 4

cost (4, 10) = min {c (10, 12) + cost (5, 12)} = 2

Therefore, cost (3, 6) = min {6 + 4, 5 + 2} = 7

---

cost (3, 7) = min {c (7, 9) + cost (4, 9) , c (7, 10) + cost (4, 10)}
= min {4 + cost (4, 9), 3 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0} = 4

Cost (4, 10) = min {c (10, 2) + cost (5, 12)} = min {2 + 0} = 2

Therefore, cost (3, 7) = min {4 + 4, 3 + 2} = min {8, 5} = 5

---

cost (3, 8) = min {c (8, 10) + cost (4, 10), c (8, 11) + cost (4, 11)}
= min {5 + cost (4, 10), 6 + cost (4 + 11)}

cost (4, 11) = min {c (11, 12) + cost (5, 12)} = 5

Therefore, cost (3, 8) = min {5 + 2, 6 + 5} = min {7, 11} = 7

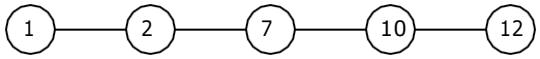
Therefore, cost (2, 2) = min {4 + 7, 2 + 5, 1 + 7} = min {11, 7, 8} = 7

Therefore, cost (2, 3) = min {c (3, 6) + cost (3, 6), c (3, 7) + cost (3, 7)}
= min {2 + cost (3, 6), 7 + cost (3, 7)}
= min {2 + 7, 7 + 5} = min {9, 12} = 9

cost (2, 4) = min {c (4, 8) + cost (3, 8)} = min {11 + 7} = 18
cost (2, 5) = min {c (5, 7) + cost (3, 7), c (5, 8) + cost (3,8)}
= min {11 + 5, 8 + 7} = min {16, 15} = 15

Therefore, cost (1, 1) = min {9 + 7, 7 + 9, 3 + 18, 2 + 15}
= min {16, 16, 21, 17} = 16

The minimum cost path is 16.

The path is 

or

BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t to s:

$$Bcost(i, J) = \min_{\substack{l \in V_{i-1} \\ <l, j> \in E}} \{Bcost(i - 1, l) + c(l, J)\}$$

Bcost (5, 12) = min {Bcost (4, 9) + c (9, 12), Bcost (4, 10) + c (10, 12),
Bcost (4, 11) + c (11, 12)}
= min {Bcost (4, 9) + 4, Bcost (4, 10) + 2, Bcost (4, 11) + 5}

Bcost (4, 9) = min {Bcost (3, 6) + c (6, 9), Bcost (3, 7) + c (7,9)}
= min {Bcost (3, 6) + 6, Bcost (3, 7) + 4}

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3,6)}
= min {Bcost (2, 2) + 4, Bcost (2, 3) + 2}

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 9} =9

Bcost (2, 3) = min {Bcost (1, 1) + c (1, 3)} = min {0 + 7} =7

Bcost (3, 6) = min {9 + 4, 7 + 2} = min {13, 9} =9

Bcost (3, 7) = min {Bcost (2, 2) + c (2, 7), Bcost (2, 3) + c (3, 7),
Bcost (2, 5) + c (5, 7)}

Bcost (2, 5) = min {Bcost (1, 1) + c (1, 5)} = 2

Bcost (3, 7) = min {9 + 2, 7 + 7, 2 + 11} = min {11, 14, 13} =11

Bcost (4, 9) = min {9 + 6, 11 + 4} = min {15, 15} = 15

Bcost (4, 10) = min {Bcost (3, 6) + c (6, 10), Bcost (3, 7) + c (7,10),
Bcost (3, 8) + c (8, 10)}

Bcost (3, 8) = min {Bcost (2, 2) + c (2, 8), Bcost (2, 4) + c (4, 8),
Bcost (2, 5) + c (5, 8)}
Bcost (2, 4) = min {Bcost (1, 1) + c (1, 4)} = 3

Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} = 10

Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14

Bcost (4, 11) = min {Bcost (3, 8) + c (8, 11)} = min {Bcost (3, 8) + 6}
= min {10 + 6} = 16

Bcost (5, 12) = min {15 + 4, 14 + 2, 16 + 5} = min {19, 16, 21} = 16.

4.3. ALL PAIRS SHORTEST PATHS

- In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G.
- That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i.
- These two paths are the same when G is undirected.
- When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.
- The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j.
- The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n^2) time, the matrix A can be obtained in O (n^3) time.
- The dynamic programming solution, called Floyd's algorithm, runs in O (n^3) time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).
- The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j.
- If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively.
- Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let A^k (i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

$$A^k(i, j) = \{\min_{1 \le k \le n} \{\min \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$$

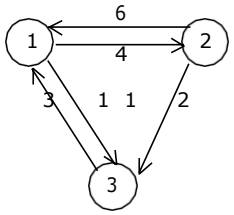
Algorithm All Paths (Cost, A, n)

```
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for 1 ≤ i ≤ n.
{
        for i := 1 to n do
                for j:= 1 to n do
                        A [i, j] := cost  [i, j];                    // copy cost into A.
        for k := 1 to n do
                for i := 1 to n do
                        for j := 1 to n do
                                A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n^3).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



Cost adjacency matrix (A^0) = $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

General formula: $\min_{1 \le k \le n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$

Solve the problem for different values of k = 1, 2 and 3

Step 1: Solving the equation for, k = 1;

$A^1(1, 1) = \min \{(A^o(1, 1) + A^o(1, 1)), c(1, 1)\} = \min \{0 + 0, 0\} = 0$

$A^1(1, 2) = \min \{(A^o(1, 1) + A^o(1, 2)), c(1, 2)\} = \min \{(0 + 4), 4\} = 4$

$A^1(1, 3) = \min \{(A^o(1, 1) + A^o(1, 3)), c(1, 3)\} = \min \{(0 + 11), 11\} = 11$

$A^1(2, 1) = \min \{(A^o(2, 1) + A^o(1, 1)), c(2, 1)\} = \min \{(6 + 0), 6\} = 6$

$A^1(2, 2) = \min \{(A^o(2, 1) + A^o(1, 2)), c(2, 2)\} = \min \{(6 + 4), 0)\} = 0$

$A^1(2, 3) = \min \{(A^o(2, 1) + A^o(1, 3)), c(2, 3)\} = \min \{(6 + 11), 2\} = 2$

$A^1(3, 1) = \min \{(A^o(3, 1) + A^o(1, 1)), c(3, 1)\} = \min \{(3 + 0), 3\} = 3$

$A^1(3, 2) = \min \{(A^o(3, 1) + A^o(1, 2)), c(3, 2)\} = \min \{(3 + 4), \infty\} = 7$

$A^1(3, 3) = \min \{(A^o(3, 1) + A^o(1, 3)), c(3, 3)\} = \min \{(3 + 11), 0\} = 0$

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solving the equation for, K = 2;

$A^2(1, 1) = \min \{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min \{(4 + 6), 0\} = 0$

$A^2(1, 2) = \min \{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min \{(4 + 0), 4\} = 4$

$A^2(1, 3) = \min \{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min \{(4 + 2), 11\} = 6$

$A^2(2, 1) = \min \{(A(2, 2) + A(2, 1), c(2, 1)\} = \min \{(0 + 6), 6\} = 6$

$A^2(2, 2) = \min \{(A(2, 2) + A(2, 2), c(2, 2)\} = \min \{(0 + 0), 0\} = 0$

$A^2(2, 3) = \min \{(A(2, 2) + A(2, 3), c(2, 3)\} = \min \{(0 + 2), 2\} = 2$

$A^2(3, 1) = \min \{(A(3, 2) + A(2, 1), c(3, 1)\} = \min \{(7 + 6), 3\} = 3$

$A^2(3, 2) = \min \{(A(3, 2) + A(2, 2), c(3, 2)\} = \min \{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min \{(A(3, 2) + A(2, 3), c(3, 3)\} = \min \{(7 + 2), 0\} = 0$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for, k = 3;

$A^3(1, 1) = \min \{A^2(1, 3) + A^2(3, 1), c(1, 1)\} = \min \{(6 + 3), 0\} = 0$

$A^3(1, 2) = \min \{A^2(1, 3) + A^2(3, 2), c(1, 2)\} = \min \{(6 + 7), 4\} = 4$

$A^3(1, 3) = \min \{A^2(1, 3) + A^2(3, 3), c(1, 3)\} = \min \{(6 + 0), 6\} = 6$

$A^3(2, 1) = \min \{A^2(2, 3) + A^2(3, 1), c(2, 1)\} = \min \{(2 + 3), 6\} = 5$

$A^3(2, 2) = \min \{A^2(2, 3) + A^2(3, 2), c(2, 2)\} = \min \{(2 + 7), 0\} = 0$

$A^3(2, 3) = \min \{A^2(2, 3) + A^2(3, 3), c(2, 3)\} = \min \{(2 + 0), 2\} = 2$

$A^3(3, 1) = \min \{A^2(3, 3) + A^2(3, 1), c(3, 1)\} = \min \{(0 + 3), 3\} = 3$

$A^3(3, 2) = \min \{A^2(3, 3) + A^2(3, 2), c(3, 2)\} = \min \{(0 + 7), 7\} = 7$

$A^3(3, 3) = \min \{A^2(3, 3) + A^2(3, 3), c(3, 3)\} = \min \{(0 + 0), 0\} = 0$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

4.5 STRING EDITING

4.5 STRING EDITING

We are given two strings $X=x_1,x_2,\ldots,x_n$ and $Y=y_1,y_2,\ldots,y_m$, where xi, $1 \le i \le n$, and y_j, $1 \le j \le m$, are members of a finite set of symbols known as the opeartions on X. The permissible edit operations are insert, delete, and change there is a cost associated with performing each. The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence. The problem of string editing is to identify a minimum cost sequence of edit operations that will transform X into Y.

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_i)$ be the cost of inserting the symbol y_j into X, and $C(x_i,y_j)$ be the cost of changing the symbol x_i of X into y_j.

Example :

Consider the string editing problem, $X = a,a,b,a,b$ and $Y=b,a,b,b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i=0$, $j>1$, and $j=0,i>1$, $cost(i,j)$ can be computed first. Let us compute the rest of the entires in row-major order. The next entry to be computed is $cost(1,1)$.

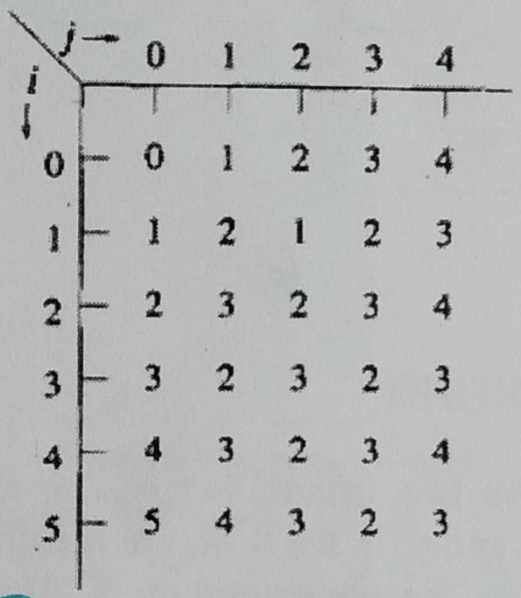
$$cost(1,1) = min \{cost(0,1)+D(x_1), cost(0,0)+ C(x_1,y_1),cost(1,0)+I(y)\}$$

$$= min \{2,2,2\} = 2$$

Next is computed cost(1,2)

$cost\ (1,2) = min\ \{cost\ (0,2) + D(x_1),\ cost(0,1)+C(x_1,y_2),\ cost(1,1) + I(y_2)\}$

$= min\ \{3,1,3\}=1$



i \ j	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

Example:

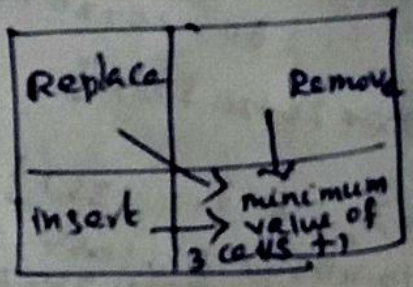
Consider the string editing problem,

x: a d c e g y = a b c f g.

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1	0	1	2	3	4
d	2	1	1	2	3	4
c	3	2	2	1	2	3
e	4	3	3	2	2	3
g	5	4	4	3	3	2

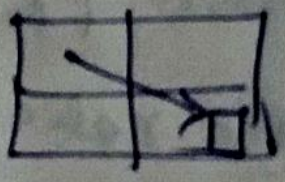
① insert → Remove ↓

② If $r \neq c$



③ If $r == c$

Just copy the diangal elements



ALGORITHM.

```
for (i = 0; i < len(str1)+1; i++)
{
  for(j=0; j < len(str2)+1; j++)
  {
    if (i==0 && j==0)
        T[i][j] = 0;
    else if(i==0)
        T[i][j] = T[i][j-1]+1;
    else if (j==0)
        T[i][j] = T[i-1][j]+1;
    else
    {
        if (str[i-1] == str2[j-1])
            T[i][j] = T[i-1][j-1];
        else T[i][j] = 1 + min( T[i][j-1]   insert
                                T[i-1][j]   remove
                                T[i-1][j-1] replace )
```

4.6. 0/1 – KNAPSACK

- We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i.
- The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.
- A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$.
- A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order x_n, x_{n-1}, x_1. Following a decision on x_n,
- we may be in one of two possible states:
  - ✓ The capacity remaining in $m - w_n$ and
  - ✓ A profit of p_n has accrued.
- It is clear that the remaining decisions $x_{n-1}, , x_1$ must be optimal with respect to the problem state resulting from the decision on x_n.
- Otherwise, $x_n, , x_1$ will not be optimal. Hence, the principal of optimality holds.

$$F_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad -- \quad 1$$

For arbitrary $f_i(y)$, $i > 0$, this equation generalizes to:

$$F_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad -- \quad 2$$

- Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_o(y) = 0$ for all y and $f_i(y) = -\infty$, $y < 0$. Then $f_1, f_2, \ldots f_n$ can be successively computed using equation–2.

- When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly.
  - Since each f_i can be computed from $f_i - 1$ in $\Theta(m)$ time, it takes $\Theta(m\ n)$ time to compute f_n.
  - When the wi's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range.
  - Even when the w_i's are integer, the explicit $\Theta(m\ n)$ computation of f_n may not be the most efficient computation.
  - So, we explore **an alternative method for both cases.**

- The $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S_i by first computing:

$$S^i_1 = \{(P, W) \mid (P - p_i, W - w_i) \in S^i\}$$

- Now, S^{i+1} can be computed by merging the pairs in S^i and S^i to1gether. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2.
- Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Algorithm :

```
Algorithm DKP(p,w,n,m)
{
S0:={(0,0)};
for i:= 1 to n-1 do
{
S_1^{i-1} : = {(P,W)|(p-p_i, W-w_i)∈S^{i-1} and W≤m};
S^i : = MergePurge (S^{i-1},S_1^{i-1});
}
(PX,WX):=last pair in S^{n-1};
(PY,WY) :=(P'+p_n, W'+w_n) where W' is the largest W in any pair in
S^{n-1} such that W+w_n≤m;
// Trace back for x_n,x_{n-1},....,x_1
if (PX>PY) then x_n:=0;
else x_n:=1;
TraceBackFor(x_{n-1},....,x_1);
}
```

Example 1:

Consider the knapsack instance n = 3, (w_1, w_2, w_3) = (2, 3, 4), (P_1, P_2, P_3) = (1,2, 5) and M = 6.

Solution:

Initially, $f_o(x) = 0$, for all x and $f_i(x) = -\infty$ if $x < 0$.

$F_n(M) = \max \{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\}$

$F_3(6) = \max (f_2(6), f_2(6 - 4) + 5\} = \max \{f_2(6), f_2(2) + 5\}$

$F_2(6) = \max (f_1(6), f_1(6 - 3) + 2\} = \max \{f_1(6), f_1(3) + 2\}$

$F_1(6) = \max (f_0(6), f_0(6 - 2) + 1\} = \max \{0, 0 + 1\} = 1$

$F_1(3) = \max (f_0(3), f_0(3 - 2) + 1\} = \max \{0, 0 + 1\} = 1$

Therefore, $F_2(6) = \max (1, 1 + 2\} = 3$

$F_2(2) = \max (f_1(2), f_1(2 - 3) + 2\} = \max \{f_1(2), -\infty + 2\}$

$F_1(2) = \max (f_0(2), f_0(2 - 2) + 1\} = \max \{0, 0 + 1\} = 1$

$F_2(2) = \max \{1, -\infty + 2\} = 1$

Finally, $f_3(6) = \max \{3, 1 + 5\} = 6$

Other Solution:

For the given data we have:

$n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(P_1, P_2, P_3) = (1, 2, 5)$ and $M = 6$.

$S^0 = \{(0,0)\}$

$S_1^0 = \{(1,2)\}$

$S^1 = \{(0,0), (1,2)\}$

$S_1^1 = \{(2,3), (3,5)\}$

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$

$S_1^2 = \{(5,4), (6,6), (7,7), (8,9)\}$

$S^3 = \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6), (7,7), (8,9)\}$

By applying Dominance rule,

$S^3 = \{(0,0), (1,2), (2,3), (5,4), (6,6)\}$

From (6,6) we can infer that the maximum profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$.

① $(6,6) \in S^3$
but $(6,6) \notin S^2$ $\therefore x_3 = 1$
$(6-5)\ (6-4) = (1,2)$

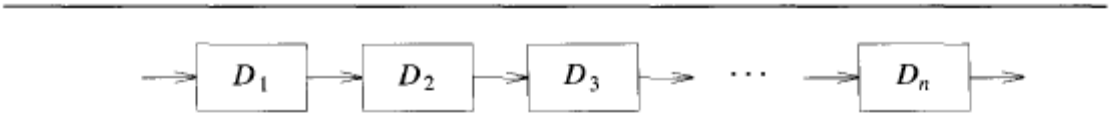
② $(1,2) \in S^2$
and $(1,2) \in S^1$ $\therefore x_2 = 0$

③ $(1,2) \in S^1$
but $(1,2) \notin S^0$ $\therefore x_1 = 1$

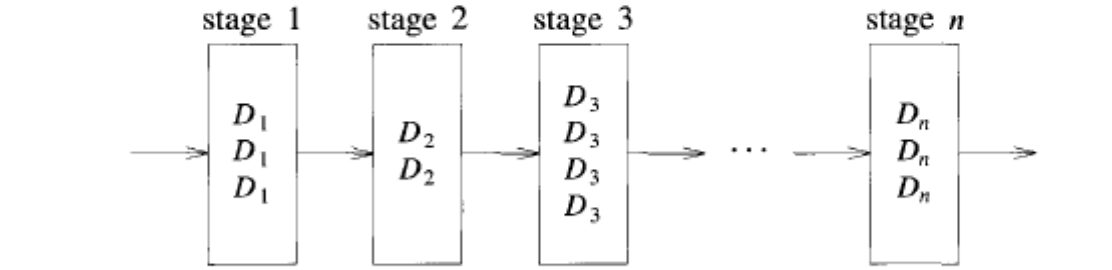
$\therefore x = \{1, 0, 1\}$

4.7. RELIABILITY DESIGN

The problem is to design a system that is composed of several devices connected in series. Let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is $\Pi\ r_i$. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, if n = 10 and r_i = 0.99, $i \leq i \leq 10$, then $\Pi\ r_i$ = .904. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.



n devices D_i, $1 \leq i \leq n$, connected in series



Multiple devices connected in parallel in each stage

If stage i contains m_i copies of device D_i. Then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$.

The reliability of stage 'i' is given by a function $\phi_i(m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed.

We wish to solve:

Maximize $\prod_{1 \leq i \leq n} \phi_i(m_i)$

Subject to $\sum_{1 \leq i \leq n} c_i m_i < C$

$m_i \geq 1$ and interger, $1 \leq i \leq n$

Assume each $C_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_J \right) \Big/ C_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$

An optimal solution $m_1, m_2 \ldots\ldots\ldots\ldots m_n$ is the result of a sequence of decisions, one decision for each m_i.

Let $f_i(x)$ represent the maximum value of $\prod_{1 \leq j \leq i} \phi(m_J)$

Subject to the constrains:

$$\sum_{1 \leq J \leq i} C_J m_J \leq x \quad \text{and } 1 \leq m_j \leq u_J,\ 1 \leq j \leq i$$

Example :

Design a three stage system with device types D_1, D_2 and D_3. The costs are \$30, \$15 and \$20 respectively. The Cost of the system is to be no more than \$105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

D_i	c_i	r_i	u_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

$C = 105$

$$u_i = \left\lfloor \left(C + c_i - \sum_{i}^{n} c_i \right) / c_i \right\rfloor$$

$$u_1 = \frac{105 + 30 - (30+15+20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30+15+20)}{15} = \frac{55}{30} = 3$$

$$u_3 = \frac{105 + 20 - (30+15+20)}{20} = \frac{60}{20} = 3$$

$S^0 = \{(1, 0)\}$

Cosider $D_1 = 1$,

$S_1^1 = \{(0.9, 30)\}$

$S_2^1 = \{(0.99, 60)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$1 - (1 - r_1)^2 =$
$= 1 - (1 - 0.9)^2$
$= 1 - (0.1)^2$
$= 1 - 0.01$
$= 0.99$,

Consider $D_2 = 2$,

$S_1^2 = \{(0.8(0.9), 15+30), (0.8(0.99), 60+15)\}$
$= \{(0.72, 45), (0.792, 75)\}$

$S_{(2)}^2 = \{(0.96(0.9), 30+15+15), (0.96(0.99), 15+15)\}$

$= \{(0.864, 60), (0.9504, 90)\}$

$S_3^2 = \{(0.992(0.9), 30+15+15+15), (0.992(0.99), 60+15+15+15)\}$

$= \{(0.8928, 75), (0.98208, 105)\}$

$1-(1-r_2)^2$	$1-(1-r_2)^3$
$= 1-(1-0.8)^2$	$= 1-(1-0.8)^3$
$= 1-(0.2)^2$	$= 1-(0.2)^3$
$= 1-0.04$	$= 1-0.008$
$= 0.96, 30$	$= 0.992, 45$

$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.9504, 90), (0.8928, 75), (0.98208, 105)\}$

By applying Dominance rule to S^2,

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Consider $D_3 = 2$,

$1-(1-r_3)^2$	$1-(1-r_3)^3$
$= 1-(1-0.5)^2$	$= 1-(1-0.5)^3$
$= 1-(0.5)^2$	$= 1-(0.5)^3$
$= 1-0.25$	$= 0.875$
$= 0.75$	

$S_1^3 = \{(0.5(0.72), 45+20), (0.5(0.864), 60+20), (0.5(0.8928), 75+20)\}$

$= \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

$S_2^3 = \{(0.75(0.72), 45+20+20), (0.75(0.864), 60+20+20), (0.75(0.8928), 75+20+20)\}$

$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

$S_3^3 = \{(0.875(0.72), 45+20+20+20), (0.875(0.864), 60+20+20+20), (0.875(0.8928), 75+20+20+20)\}$

$= \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$

$S^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95), (0.54, 85), (0.6696, 115), (0.63, 105), (1.756, 120), (0.648, 100), (0.7812, 135)\}$

If cost exceeds 105, remove that tuples,

$S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$

The best design has a reliability of 0.648 and a cost of 100. D^i's we can determine that $D_3 = 2$, $D_2 = 2$ and $D_1 = 1$

4.8. TRAVELLING SALESPERSON PROBLEM

- Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \alpha$ if < i, j> $\notin$ E. Let |V| = n and assume n > 1.
- A tour of G is a directed simple cycle that includes every vertex in V.
- The cost of a tour is the sum of the cost of the edges on the tour.
- The traveling sales person problem is to find a tour of minimum cost.
- The tour is to be a simple path that starts and ends at vertex 1.
- Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1.
- The function g (1, V – {1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{--} \quad 1$$

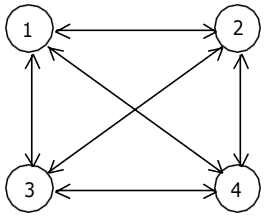
Generalizing equation 1, we obtain (for i $\notin$ S)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(i, S - \{j\})\} \quad \text{--} \quad 2$$

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

Example :

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, K\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \quad - \quad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \le i \le n$. So,

$g(2, \Phi) = C_{21} = 5$

$g(3, \Phi) = C_{31} = 6$ g (4,

$\Phi) = C_{41} = 8$

Using equation – (2) we obtain:

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$

$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$
$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$

$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\}), (c_{34} + g(4,\{2\})\}$

$g(2, \{4\}) = \min \{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$g(2, \{3\}) = \min \{c_{23} + g(3, \Phi\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c_{32} + g(2, \Phi\} = 13 + 5 = 18$

Therefore, $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$
$= \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35 The

optimal tour is: 1, 2, 4, 3, 1.

4.9. TECHNIQUES FOR BINARY TREES

- **A binary tree** is a finite collection of elements or it can be said it is made up of nodes. Where each node contains the left pointer, right pointer, and a data element.
- The root pointer points to the topmost node in the tree.
- When the binary tree is not empty, so it will have a root element and the remaining elements are partitioned into two binary trees which are called the left pointer and right pointer of a tree.

Traversing in the Binary Tree:

- **Tree traversal** is the process of visiting each node in the tree exactly once.
- Visiting each node in a graph should be done in a systematic manner.
- If search result in a visit to all the vertices, it is called a traversal.
- There are basically three traversal techniques for a binary tree that are,
    1. Preorder traversal
    2. Inorder traversal
    3. Postorder traversal

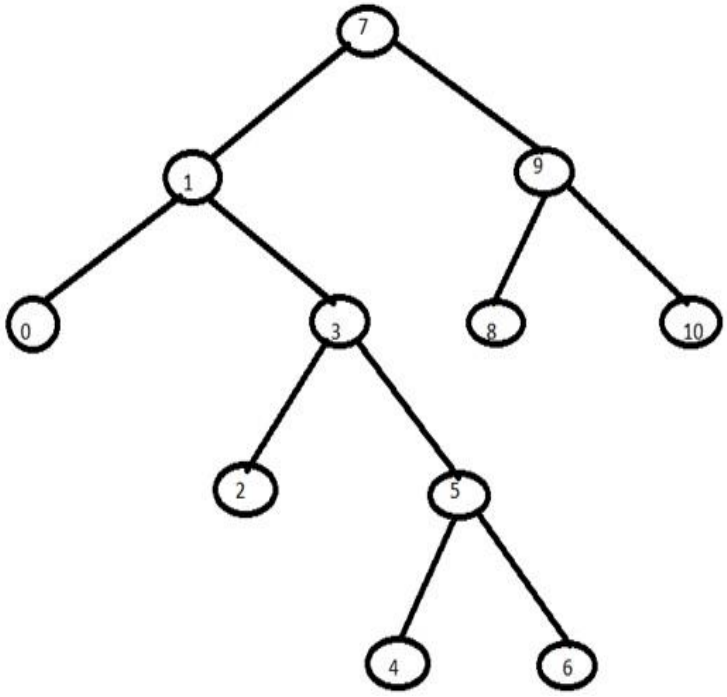
1) Preorder traversal

- This technique follows the 'root left right' policy.
- It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed.
- As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.
- So, in a preorder traversal, each node is visited before both of its subtrees.
- To **traverse a binary tree in preorder**, following operations are carried out:
    1. Visit the root.
    2. Traverse the left sub tree of root.
    3. Traverse the right sub tree of root.

Algorithm:

```
Algorithm preorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
  If t! =0 then
  {
            Visit(t);
            Preorder(t->lchild);
            Preorder(t->rchild);
  }
}
```

Example: Let us consider the given binary tree,



Therefore, the preorder traversal of the above tree will be: **7,1,0,3,2,5,4,6,9,8,10**

The applications of preorder traversal include:

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

2) Inorder traversal:

- This technique follows the 'left root right' policy.
- It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed.
- As the root node is traversed between the left and right subtree, it is named inorder traversal.
- So, in the inorder traversal, each node is visited in between of its subtrees.
- To traverse a binary tree in inorder traversal, following operations are carried out:
    1. Traverse the left most sub tree.
    2. Visit the root.
    3. Traverse the right most sub tree.

Note: Inorder traversal is also known as LNR traversal.

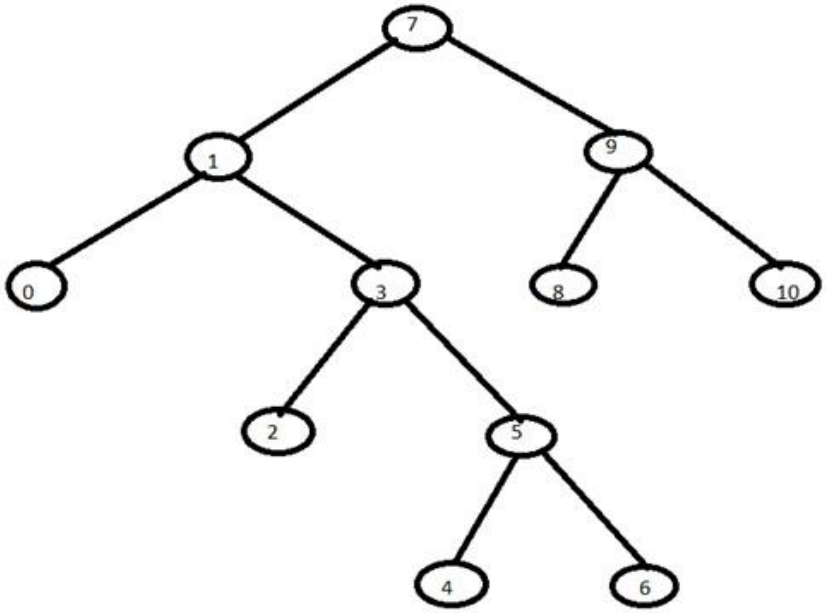
The applications of Inorder traversal includes :

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

Algorithm:

```
lgorithm inorder(t)

/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Inorder(t->lchild);
                Visit(t);
                Inorder(t->rchild);
        }
}
```

Example: Let us consider a given binary tree.



Therefore the inorder traversal of above tree will be: **0,1,2,3,4,5,6,7,8,9,10**

3) Postorder traversal:

- This technique follows the 'left-right root' policy.
- It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed.
- As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.
- So, in a postorder traversal, each node is visited after both of its subtrees.
- To traverse a binary tree in postorder traversal, following operations are carried out:

1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

Note: Postorder traversal is also known as LRN traversal.

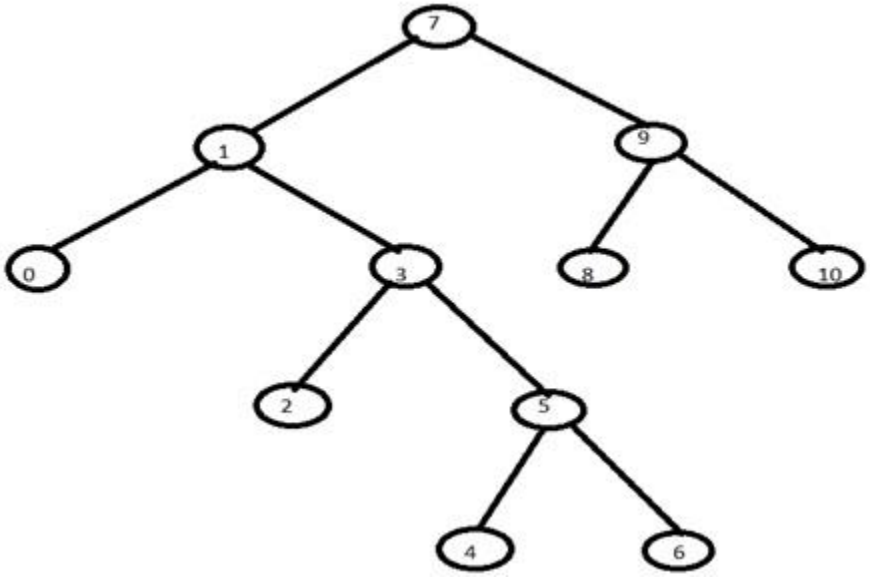
The applications of postorder traversal include:

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

Algorithm:

```
Algorithm postorder(t)

/*t is a binary tree .Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Postorder(t->lchild);
                Postorder(t->rchild);
                Visit(t);
        }
}
```

Example: Let us consider a given binary tree.



Therefore the postorder traversal of the above tree will be: **0,2,4,6,5,3,1,8,10,9,7**

4.10. TECHNIQUES FOR GRAPH

- Graph traversal is a technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

4.10.1. GRAPH TRAVERSAL - DFS

DFS (Depth First Search):

- DFS traversal of a graph produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without loops.
- We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

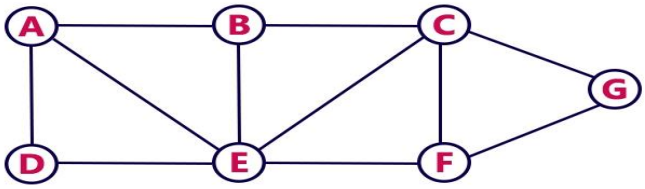
Use the following steps to implement DFS traversal...

- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

- **Back tracking** is coming back to the vertex from which we reached the current vertex.

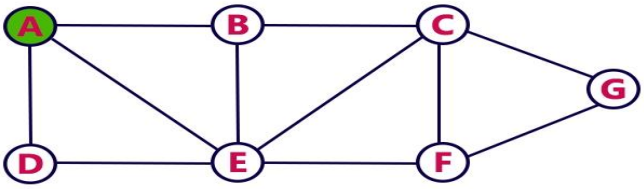
Example:

Consider the following example graph to perform DFS traversal



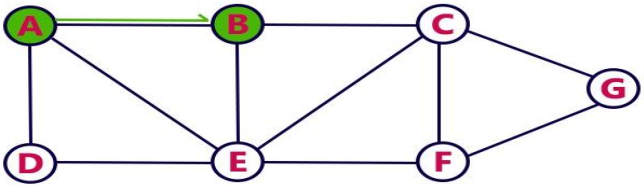
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



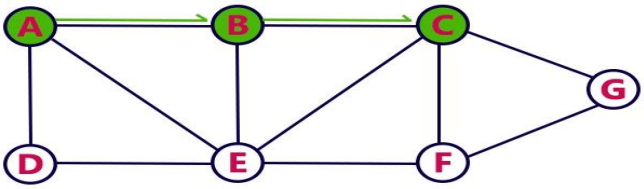
Step 2:

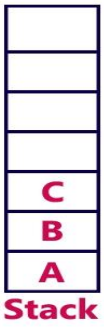
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

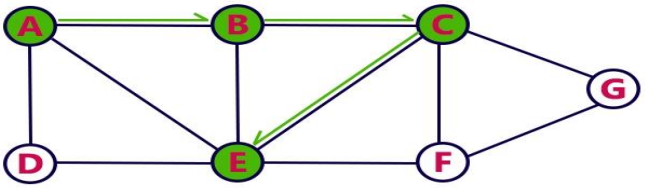
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





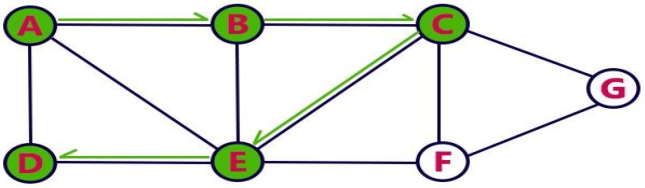
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



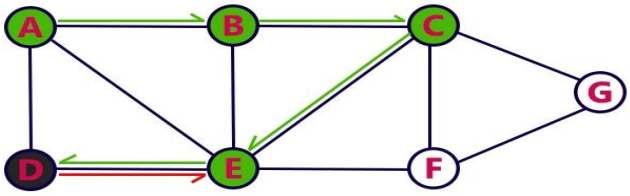
Step 5:

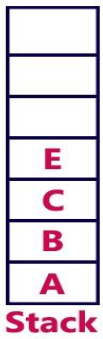
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



Step 6:

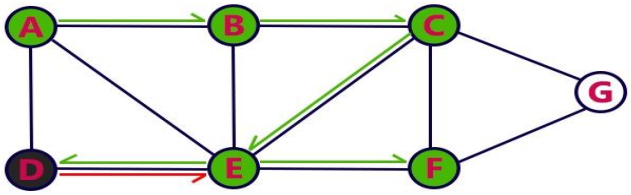
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.





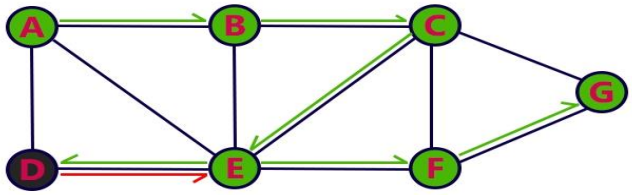
Step 7:

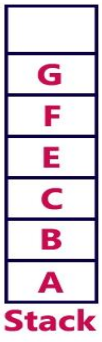
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



Step 8:

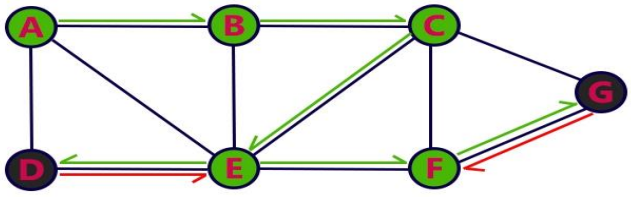
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

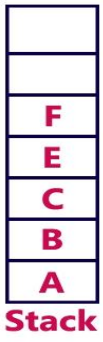




Step 9:

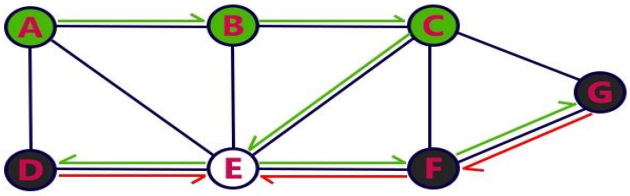
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

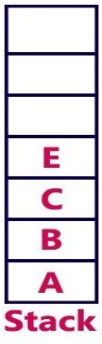




Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

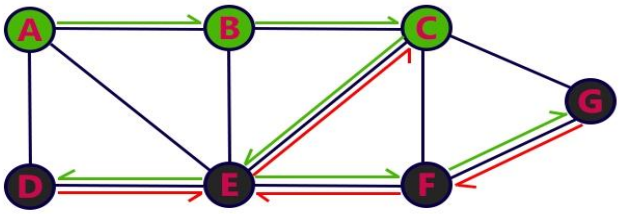


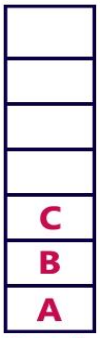


Stack

Step 11:

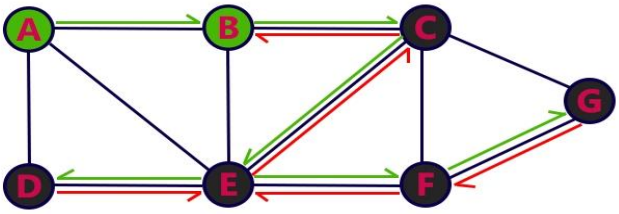
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

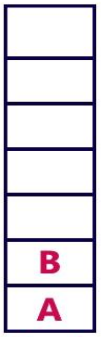




Step 12:

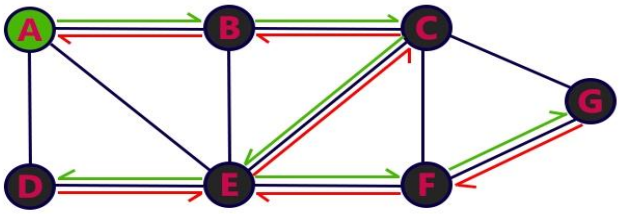
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

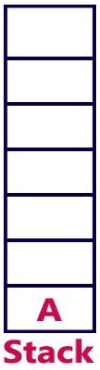




Step 13:

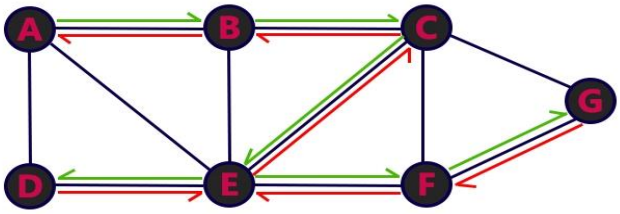
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

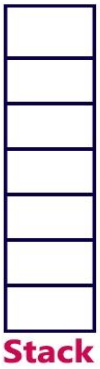




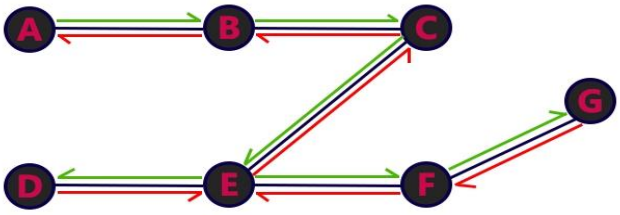
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Applications of Depth First Search:

1. Detecting cycle in a graph: A graph has a cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

2. Path Finding: We can specialize the DFS algorithm to find a path between two given vertices u and z.

- Call DFS(G, u) with u as the start vertex.
- Use a stack S to keep track of the path between the start vertex and the current vertex.
- As soon as destination vertex z is encountered, return the path as the contents of the stack

3. Topological Sorting: Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers.

4. To test if a graph is bipartite: We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.

5. Finding Strongly Connected Components of a graph: A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See **this** for DFS-based algo for finding Strongly Connected Components)

6. Solving puzzles with only one solution: such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.).

7. Web crawlers: Depth-first search can be used in the implementation of web crawlers to explore the links on a website.

8. Maze generation: Depth-first search can be used to generate random mazes.

9. Model checking: Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

10. Backtracking: Depth-first search can be used in backtracking algorithms.

Advantages of Depth First Search:

- Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- The time complexity of a depth-first Search to depth d and branching factor b (the number of children at each node, the outdegree) is O(bd) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
- DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

Disadvantages of Depth First Search:

- The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Even a finite graph can generate an infinite tre One solution to this problem is to impose a cutoff depth on the search. Although ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.
- Depth-First Search is not guaranteed to find the solution.
- And there is no guarantee to find a minimal solution, if more than one solution.

4.10.2. GRAPH TRAVERSAL - BFS

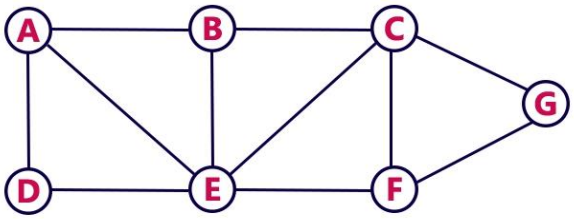
BFS (Breadth First Search):

- BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.
- We use the following steps to implement BFS traversal...

- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

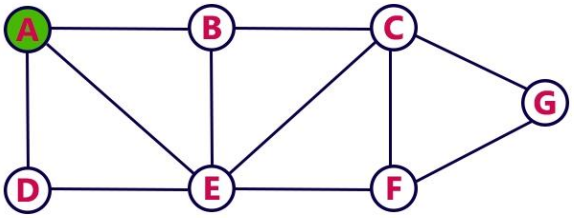
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

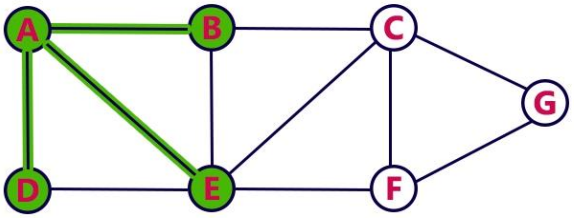


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

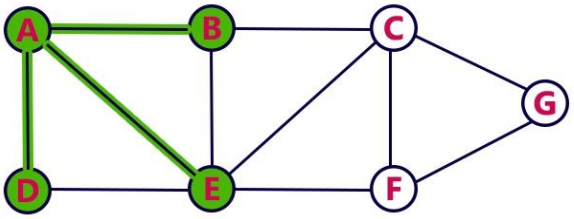


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

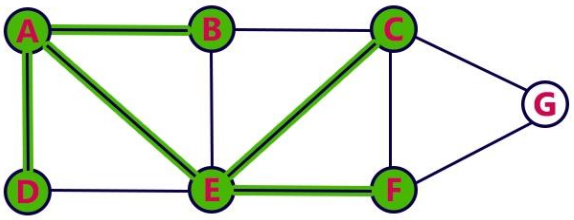


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

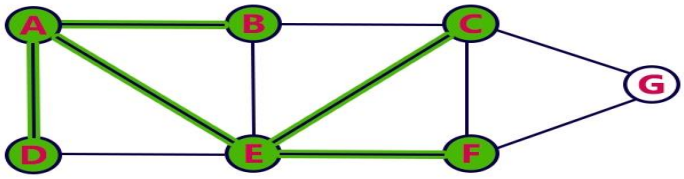


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

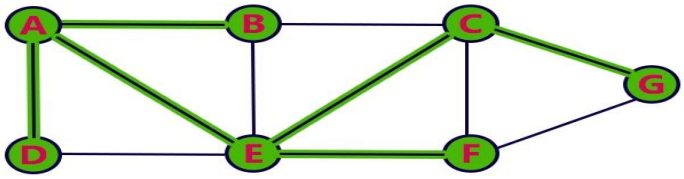


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

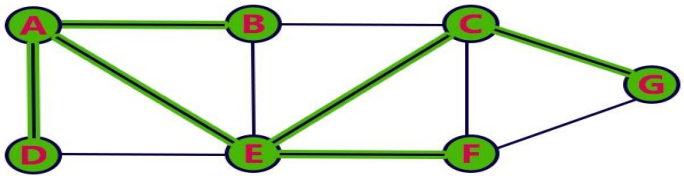


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

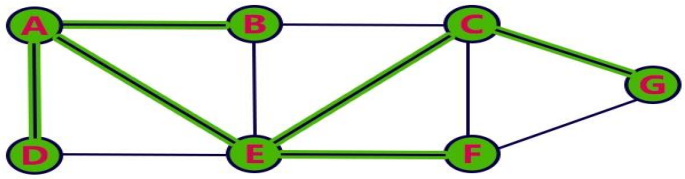


Queue

						G

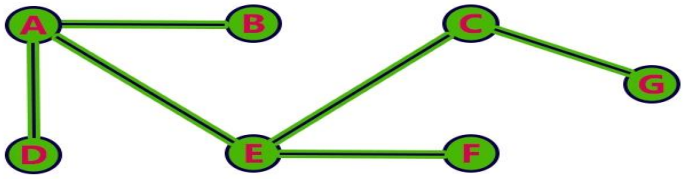
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Applications of Breadth First Search:

1. Shortest Path and Minimum Spanning Tree for unweighted graph: In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges. Also, in the case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2. Minimum Spanning Tree for weighted graphs: We can also find Minimum Spanning Tree for weighted graphs using BFT, but the condition is that the weight should be non-negative and the same for each pair of vertices.

3. Peer-to-Peer Networks: In Peer-to-Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

4. Crawlers in Search Engines: Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.

5. Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

6. GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

7. Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

8. In Garbage Collection: Breadth First Search is used in copying garbage collection using **Cheney's algorithm**. Breadth First Search is preferred over Depth First Search because of a better locality of reference.

9. Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect a cycle. We can use BFS to detect cycle in a directed graph also.

10. Ford–Fulkerson algorithm In Ford – Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces the worst-case time complexity to $O(VE^2)$.

11. To test if a graph is Bipartite: We can either use Breadth First or Depth First Traversal.

12. Path Finding: We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

13. Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

14. AI: In AI, BFS is used in traversing a game tree to find the best move.

15. Network Security: In the field of network security, BFS is used in traversing a network to find all the devices connected to it.

16. Connected Component: We can find all connected components in an undirected graph.

17. Topological sorting: BFS can be used to find a topological ordering of the nodes in a directed acyclic graph (DAG).

18. Image processing: BFS can be used to flood-fill an image with a particular color or to find connected components of pixels.

19. Recommender systems: BFS can be used to find similar items in a large dataset by traversing the items' connections in a similarity graph.

20. Other usages: Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structures similar to Breadth First Search.

Advantages of Breadth First Search:

- ✓ BFS will never get trapped exploring the useful path forever.
- ✓ If there is a solution, BFS will definitely find it.
- ✓ If there is more than one solution then BFS can find the minimal one that requires less number of steps.
- ✓ Low storage requirement – linear with depth.
- ✓ Easily programmable.

Disadvantages of Breadth First Search:

- ✓ The main drawback of BFS is its memory requirement. Since each level of the graph must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is $\mathbf{O(b^d)}$, where **b** is the branching factor(the number of children at each node, the outdegree) and **d** is the depth.
- ✓ As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT V

Problem Solving Methods: The General Method – The 8– Queens Problem – Sum of Subsets– Graph Coloring –Hamiltonian Cycles – Branch and Bound: General Method – LC Branch and Bound – FIFOBranch and Bound.

5.1. BACKTRACKING GENERAL METHOD

- Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy.
- It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions.
- This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems.
- In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.
- Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again.
- It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. The name backtrack was first coined by D. H. Lehmer in the 1950s. Early workers who studied the process were R. J. Walker, who gave an algorithmic account of it in 1960, and S. Golomb and L. Baumert who presented a very general description of it as well as a variety of applications.

In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$, where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a *criterion function* $P(x_1, \ldots, x_n)$. Sometimes it seeks all vectors that satisfy P. For example, sorting the array of integers in $a[1 : n]$ is a problem whose solution is expressible by an n-tuple, where x_i is the index in a of the ith smallest element. The criterion function P is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i < n$. The set S_i is finite and includes the integers 1 through n. Though sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an n-tuple. In this chapter we study a collection of problems whose solutions are best done using backtracking.

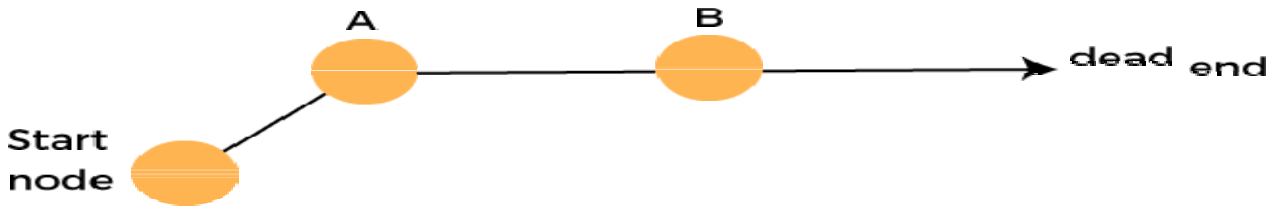
When to use a Backtracking algorithm?

- When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:
  - A piece of sufficient information is not available to make the best choice, so we use the backtrackingstrategy to try out all the possible solutions.
  - Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In thiscase, we need to use the backtracking strategy.

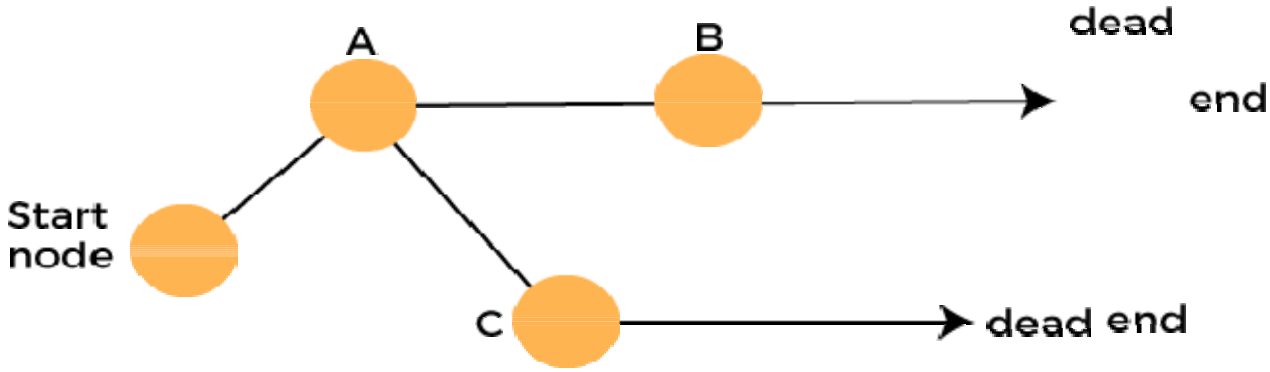
How does Backtracking work?

- Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

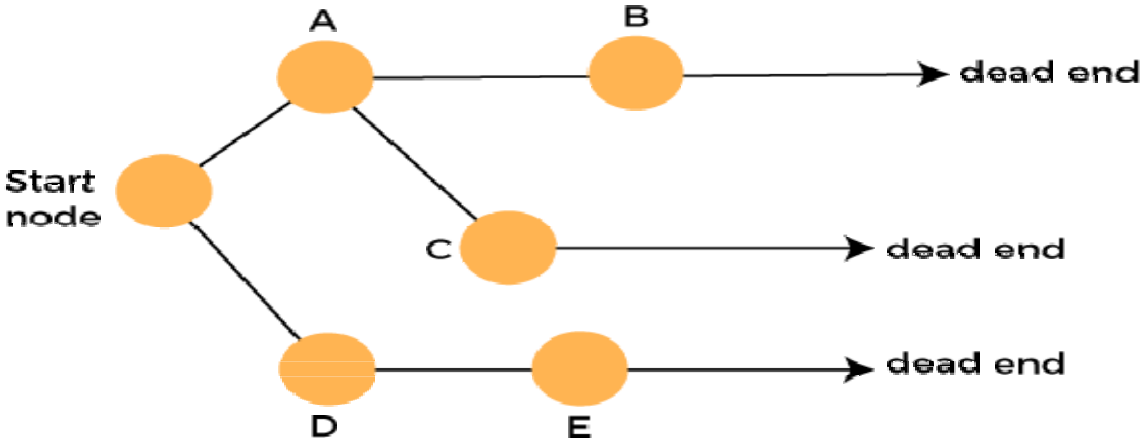
- We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



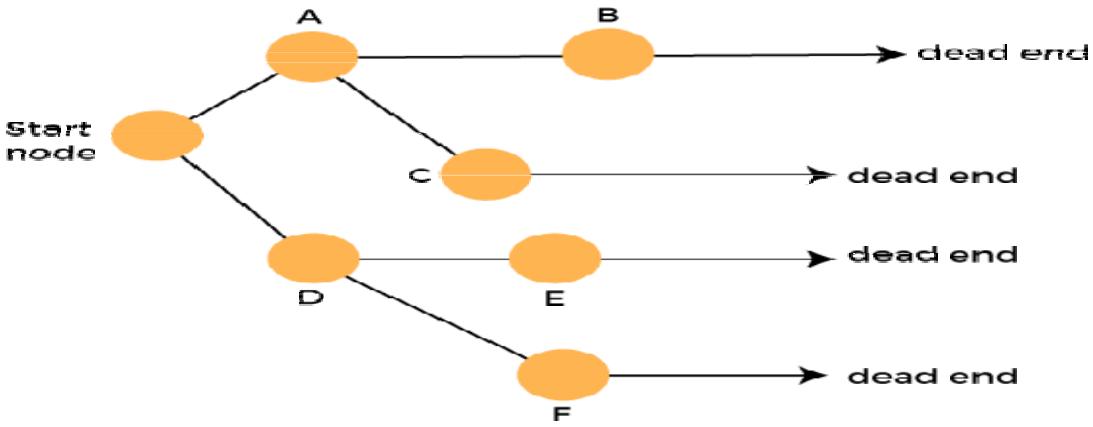
- Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead- end, so again backtrack from node C to node A. We move from node A to the starting node.



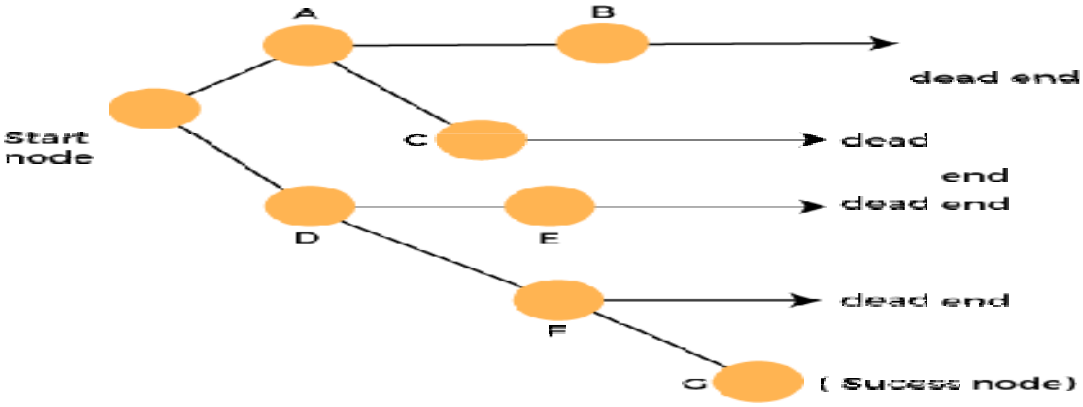
- Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



- Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



- Suppose there is another path exists from the node F to node G so move from node F to node G. The node Gis a success node.



The terms related to the backtracking are:

- **Live node:** The nodes that can be further generated are known as live nodes.
- **E node:** The nodes whose children are being generated and become a success node.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasiblesolution is known as a dead node.

- Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints,and these constraints are of two types:
  - **Implicit constraint:** It is a rule in which how each element in a tuple is related.
  - **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking:

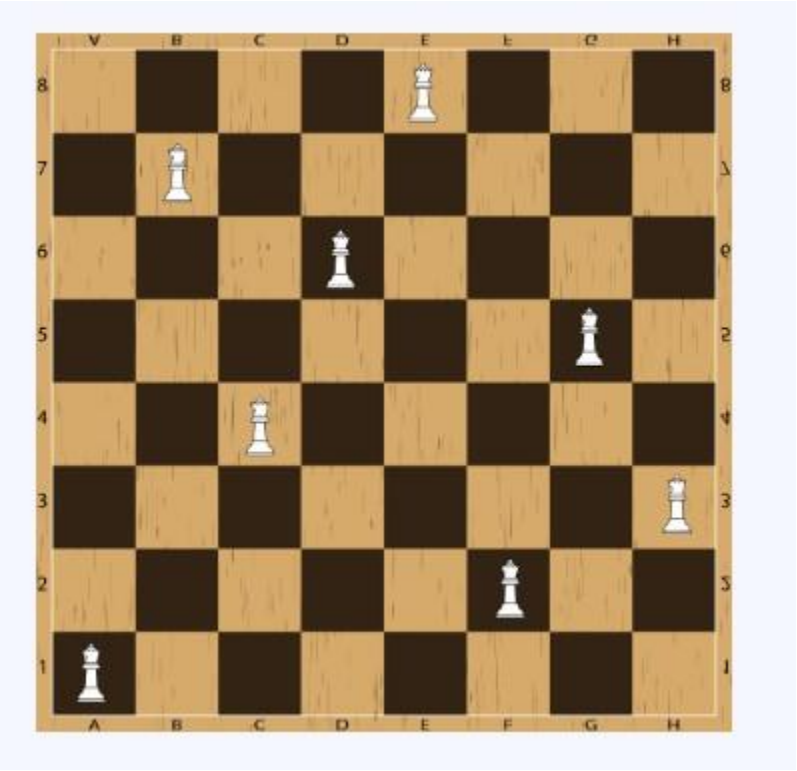
- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

Difference between the Backtracking and Recursion:

- Recursion is a technique that calls the same function again and again until you reach the base case.
- Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.

5.2. THE 8– QUEENS PROBLEM

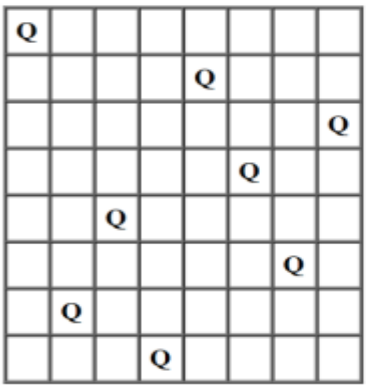
- The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal).
- More generally, the n queens problem places n queens on an n×n chessboard. There are different solutions for the problem.



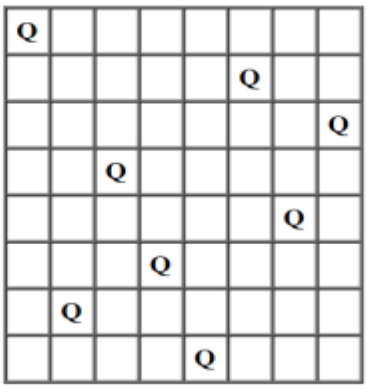
```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

Solutions:

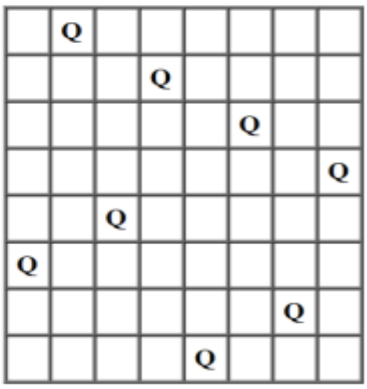
- The eight queens puzzle has 92 distinct solutions. If solutions that differ only by the symmetry operations of rotation and reflection of the board are counted as one, the puzzle has 12 solutions.
- These are called fundamental solutions; representatives of each are shown below. A fundamental solution usually has eight variants (including its original form) obtained by rotating 90, 180, or 270° and then reflecting each of the four rotational variants in a mirror in a fixed position.
- However, should a solution be equivalent to its own 90° rotation (as happens to one solution with five queens on a 5×5 board), that fundamental solution will have only two variants (itself and its reflection).
- Should a solution be equivalent to its own 180° rotation(but not to its 90° rotation), it will have four variants (itself and its reflection, its 90° rotation and the reflection of that).
- If n > 1, it is not possible for a solution to be equivalent to its own reflection because that would require two queens to be facing each other. Of the 12 fundamental solutions to the problem with eight queens on an 8×8 board, exactly one (solution 12 below) is equal to its own 180° rotation, and none is equal to its 90° rotation; thus, the number of distinct solutions is 11×8 + 1×4 = 92. All fundamental solutions are presented below



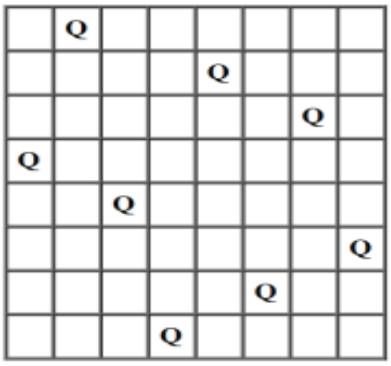
Solution 1



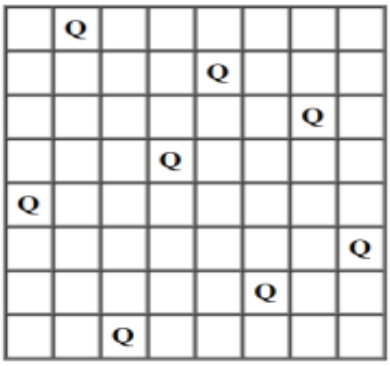
Solution 2



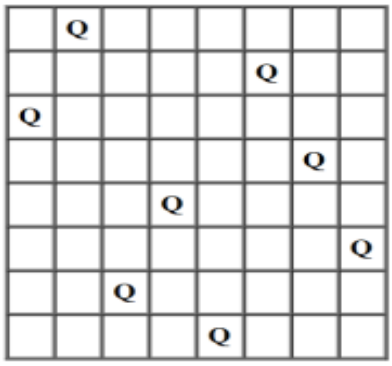
Solution 3



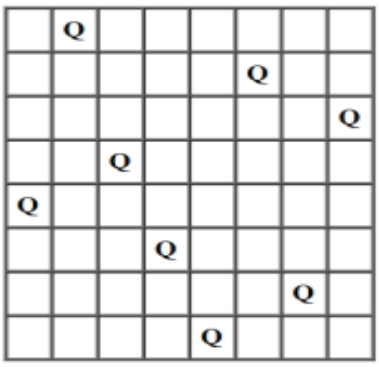
Solution 4



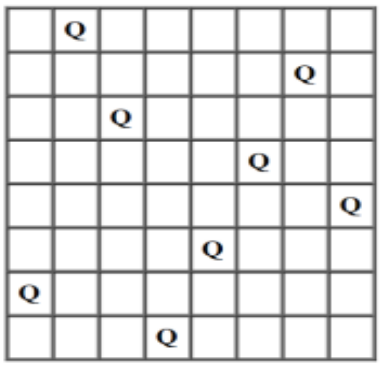
Solution 5



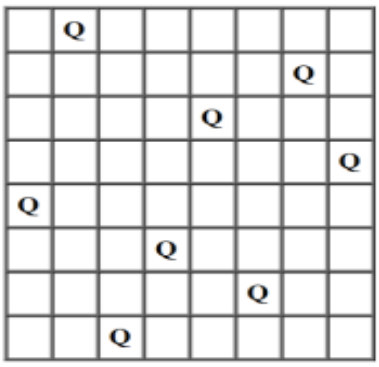
Solution 6



Solution 7



Solution 8



Solution 9

5.3. SUM OF SUBSETS

- Suppose we are given n distinct positive numbers(usually called weights) and we desire to find all combinations of these numbers whose sum are m. This is called the sum of subsets problem.
- (Example 1) given positive numbers Wi, 1<=i<=n, and m, this problem calls for finding all subsets of wi whose sums are m. For example, if n=4, (w1,w2,w3,w4)=(7,11,13,24) and m=31, then the desired subsets are (7,11,13) and (7,24). Rather than representing the solution vector by wi which sum to m, we could represent the solution vector by giving the indices of these wi.
- Now the two solutions are described by the vectors(1,2,3) and (1,4).
- In general all solution subset is represented by n-tuple(X1,X2,X3,…Xn) such that XiÎ {0,1},1<=i<=n. The Xi is 0 if wi is not chosen and xi=1 if wi is chosen. The solutions to the above instances are (1,1,1,0) and (1,0,0,1).
- This formulation expresses all solutions using a fixed sized tuple.
- The sum of sub set is based on fixed size tuple. Let us draw a tree structure for fixed tuple size formulation.
- All paths from root to a leaf node define a solution space. The left subtree of the root defines all subsets containing W1 and the right subtree defines all subsets not containing W1 and so on.

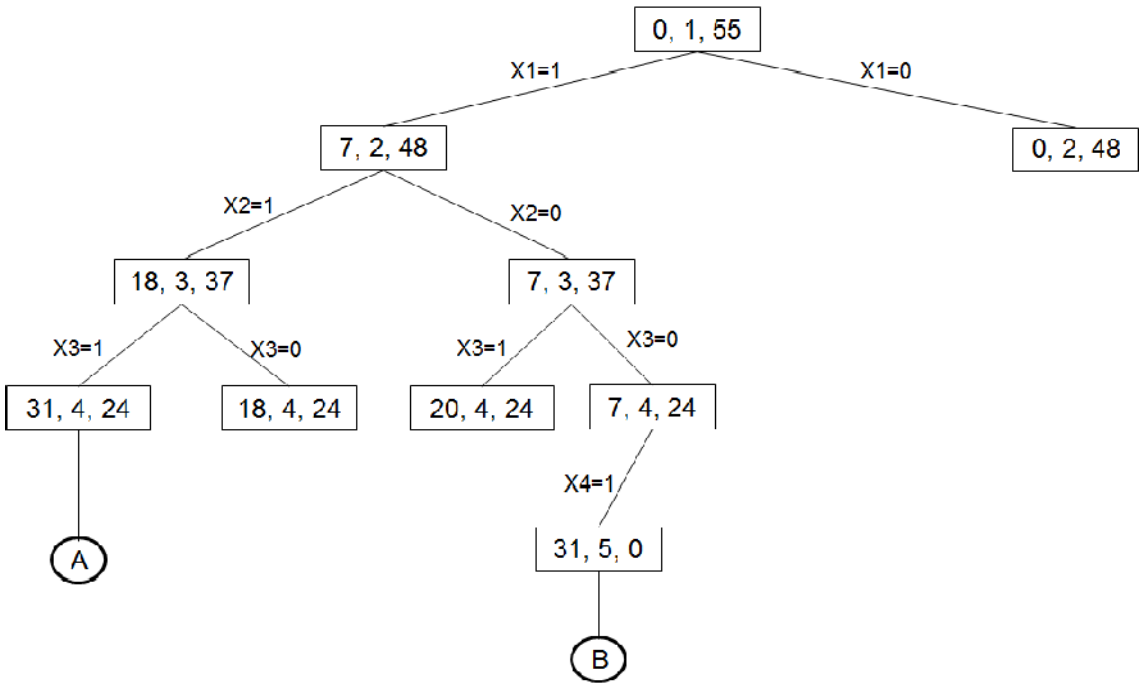
Step 1) Start with an empty set
Step 2) Add next element in the list to the sub set
Step 3)If the subset is having sum = m then stop withthat sub set as solution.
Step 4)If the sub set is not feasible or if we have reached the end of the set then backtrack through thesubset until we find the most suitable value.
Step 5) if the subset is feasible then repeat step 2
Step 6) if we have visited all elements without finding asuitable subset and if no backtracking is possible, then stop with no solution.

- s – sum of all selected elements
- k – denotes the index of chosen element
- r – initially sum of all elements. After selection of
- some element from the set subtract the chosen value from r each time.W(1:n) – represents set containing n elements.
- X[i]-solution vector 1<=i<=k

Algorithm for sum of subsets:

```
Algorithm sumofsubsets(s,k,r)
{
X[k]:=1;
if (s+w[k]=m) then write (x[1:k]); // subset foundelse
if (s+w[k]+w[k+1]<=m) then
sumofsubsets(s+w[k],k+1,r-w[k]);
//generate right child and evaluate Bk.
if ((s+r-w[k]>=m) and (s+w[k+1]<=m)) then
{
X[k]:=0;
sumofsubsets(s,k+1,r-w[k]);
}
}
```

Example:
n=4, (w1,w2,w3,w4)=(7,11,13,24) and m=31
Solution Vector=(x[1],x[2],x[3],x[4])



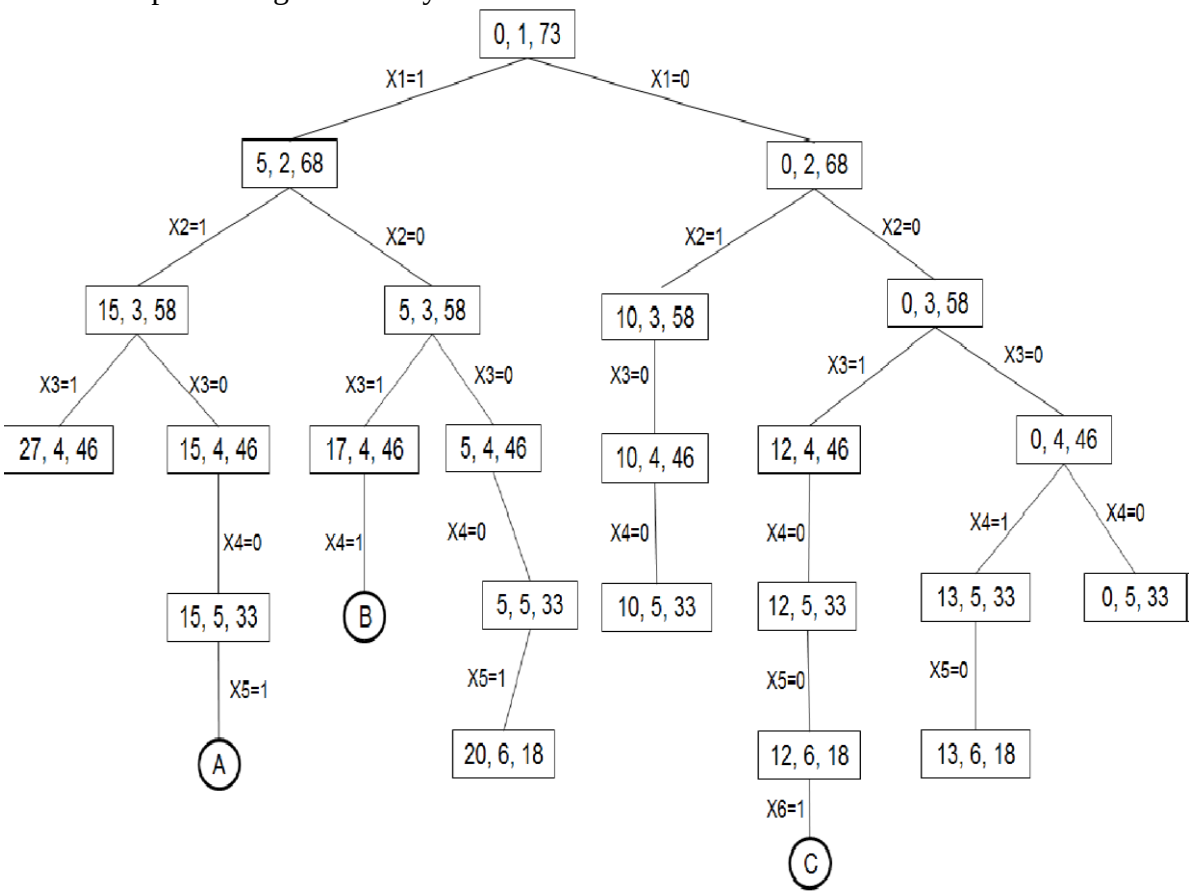
Portion of state space Tree

Solution A = {1,1,1,0}
Solution B = {1,0,0,1}

Example 2:

n=6, m=30 and w[1:6]={5,10,12,13,15,18}.

Portion of the state space tree generated by sum of subsets



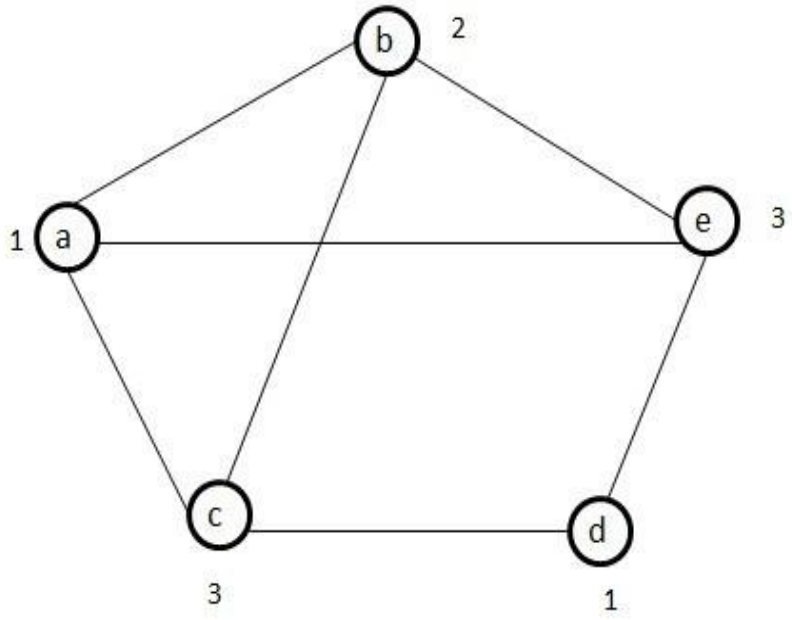
State space tree with solution
The rectangular nodes list the values of s,k and r.
Circular nodes represent points at which subsets with
sums m are printed out.

Solution A = (1,1,0,0,1)
Solution B = (1,0,1,1)
Solution C = (0,0,1,0,0,1)

Note that the tree contains only 23 rectangular nodes.
The full space tree for n=6 contains $2^6-1=63$ nodes from which calls could be made.

5.4. GRAPH COLORING

- Let G be a graph and m be a given positive integer. We want to discover whether the node of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.
- This is termed the m-colorability decision problem. Note that if d is the degree of the given graph, then it canbe colored with d+1 colors. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. The integer is referred to as the chromatic number of the graph.
- For example the following graph can be colored with three colors 1, 2 and 3.
- The color of each node is indicated next to it. It can also be seen that thee colors are needed to color thisgraph and hence this graph's chromatic number is 3.



An example graph and its coloring

State space tree for coloring a graph containing 3 nodes using 3 colors

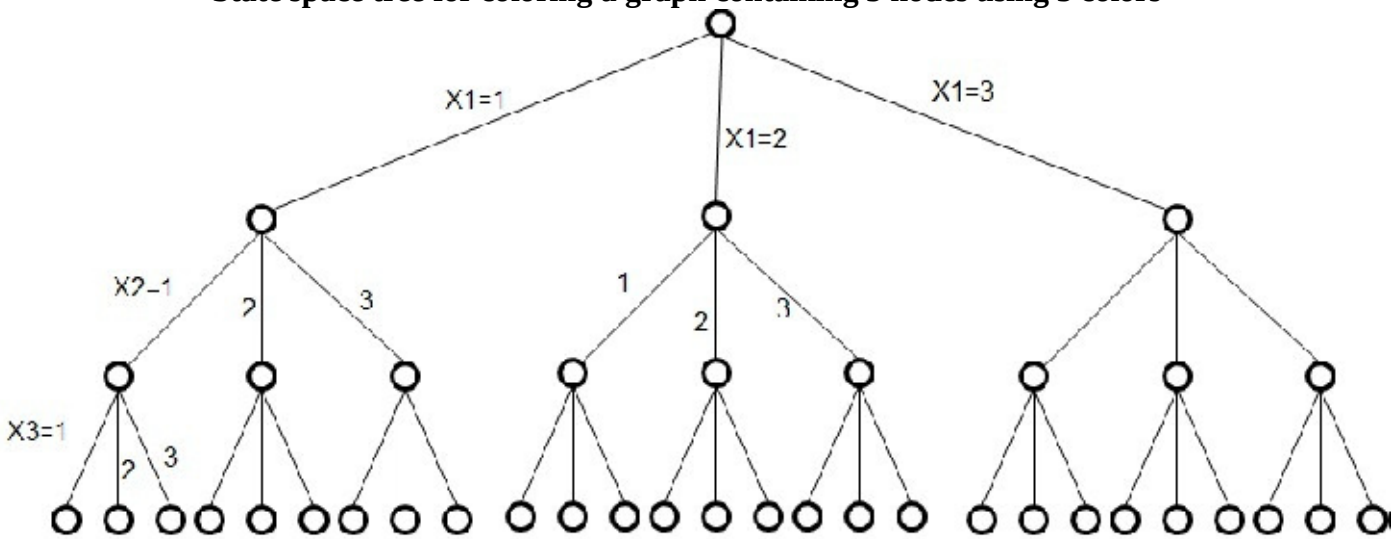


Fig)State space tree for mColoring when n=3 and m=3

- The algorithm mcoloring was formed using the recursive backtracking schema.
- The graph is represented by its Boolean adjacency matrix G[1:n,1:n].
- All assignments of 1,2,…m to the vertices of the graph such that adjacent vertices are assigned distinct integers are printed. K is the index of the next vertex to color.

Algorithm for graph coloring:

Algorithm mcoloring(k)

```
{
repeat
{
nextvalue(k);
if (x[k] = 0) then return;
if (k=n) then
write(x[1:n]);
else
mcoloring(k+1);
}until(false);
}
```

No of vertices= n
No of colors= m

- Solution vector = X[1], X[2], X[3] X[n] The values of solution vector may belongs to {0,1,2,3..m}

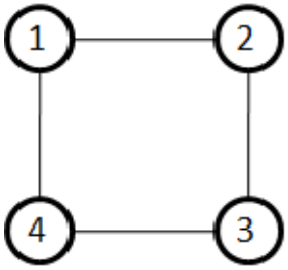
The following Algorithm is used to generate next color.

- Assume that X[1],..x[k-1] have been assigned integer values in the range [1,m] such that adjacent verticeshave distinct integers.
- A value for x[k] is determined in the range [0,m].

- X[k] is assigned the next highest numbered color while maintaining distinctness from the adjacent verticesof vertex k. if no such color exists, the x[k]=0.

Algorithm nextvalue(k)

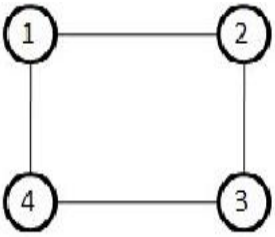
```
{
Repeat
{
X[k]=(x[k]+1)mod(m+1); // next highest color
if (x[k]=0) then
return; //all colors have been used
for j:=1 to n do
{
if ((G[k,j]!=0) and (x[k]=x[j])) then break;
//g[k,j] an edge and
//vertices k and j have same color
}
if (j=n+1) then return;
}until (false);
}
```

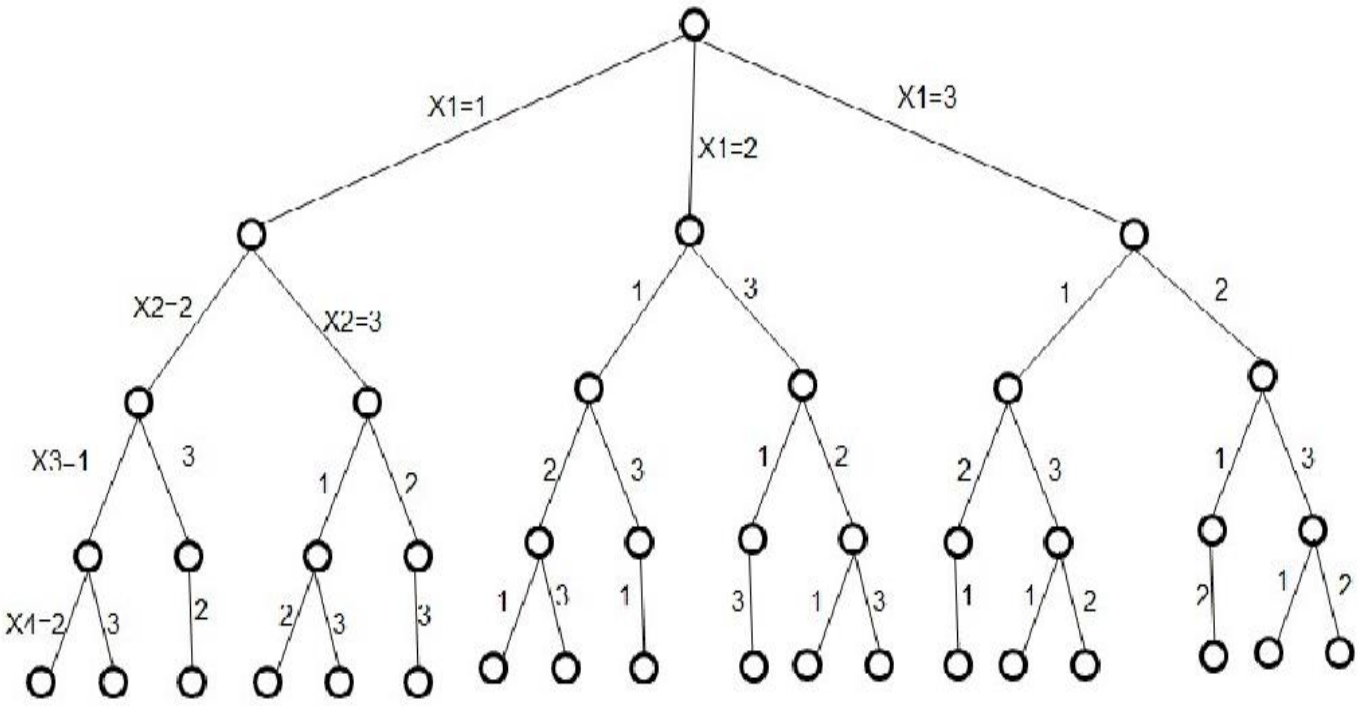


Graph

	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

Adjacency Matrix

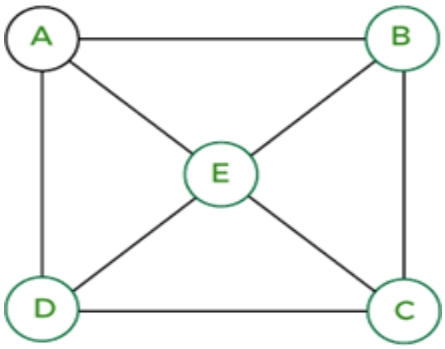




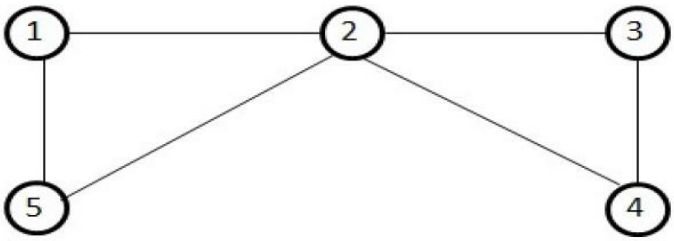
A 4-node graph and all possible 3-colorings

5.5. HAMILTONIAN CYCLES

- Let G=(V,E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex v1 G and the vertices of G are visited in the order v1, v2,…vn+1 then the edges (vi,vi+1) are in E, 1<=i<=n, and the vi are distinct except for v1 and vn+1,



which are equal.



Graph Does not containing a Hamiltonian Cycle

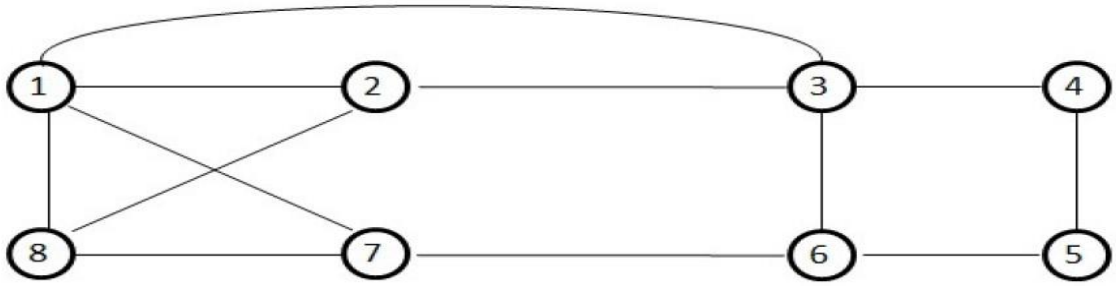
- To check whether there is a Hamiltonian cycle or not we may use backtracking method. The graph may be directed or undirected. Only distinct cycles are output.
- The backtracking solution vector (X_1,X_2,X_3,…X_n) is defined so that xi represents the ith visited vertex of theproposed cycle.
- Now all we need to do is determine how to compute the set of possible vertices for x_k if x_1,..x_{k-1} have alreadybeen chosen. If k=1 then x1 can be any of the n vertices.
- The algorithm Hamiltonian() uses the recursive formulation of backtracking to find all the Hamiltoniancycles of a graph. The graph is stored as an adjacency matrix G[1:n,1:n]. All cycles begin at node 1.

Algorithm for Hamiltonian Cycle:

```
Algorithm Hamiltonian(k)
{
Repeat
{
nextvalue(k);
if (x[k]=0) then return;
if (k=n) then write (x[1:n]);
else
Hamiltonian(k+1);
}until(false);
}
```

Algorithm to find all Hamiltonian cycles.

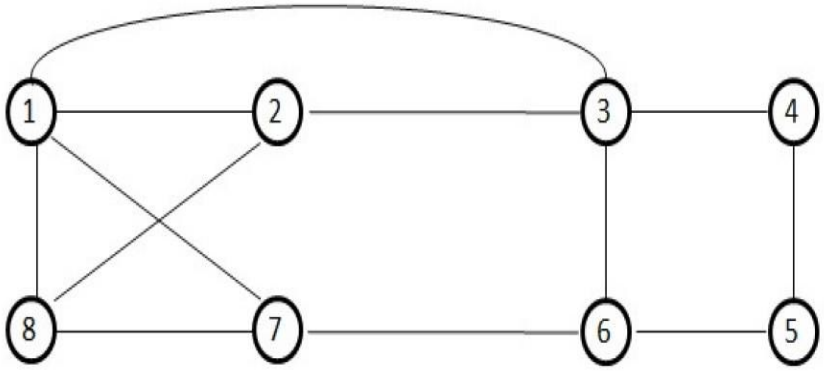
Example)



No of vertices n=8

Adjacency matrix G

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	1	1
2	1	0	1	0	0	0	0	1
3	1	1	0	1	0	1	0	0
4	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	0	0
6	0	0	1	0	1	0	1	0
7	1	0	0	0	0	1	0	1
8	1	1	0	0	0	0	1	0




```
The solution vector for hamiltonian cycles
1,2,3,4,5,6,7,8,1
1,8,2,3,4,5,6,7,1
1,3,4,5,6,7,8,2,1
```

5.6. BRANCH AND BOUND

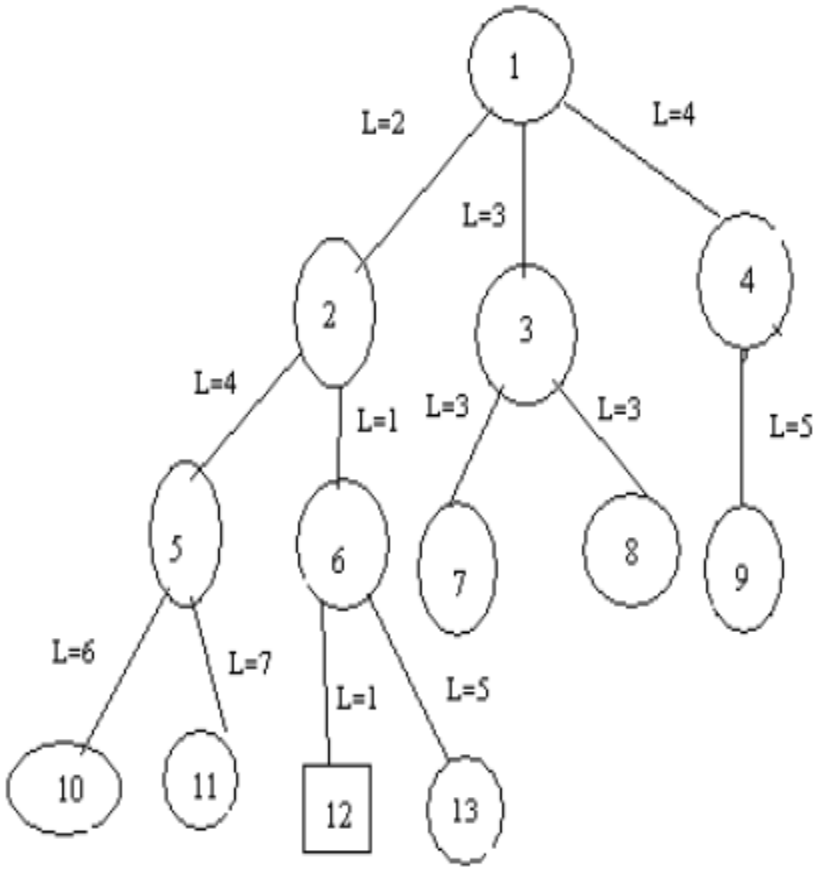
Introduction:

- Branch and Bound refers to all state space search methods in which all children of the Enode are generated before any other live node becomes the E-Node. Branch and Bound is the generalization of both graph search strategies, BFS and D-search.
- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out. A D-search like state space search is called as LIFO (last in first out) search as the list of live nodes in a last in first out list.
- Live node is a node that has been generated but whose children have not yet been generated.
- E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- Dead node is a generated anode that is not be expanded or explored any further. All children of a dead node have already been expanded.
- 3 types of search strategies:

1. FIFO (First In First Out)
2. LIFO (Last In First Out)
3. LC (Least Cost)
4.

5.7. LC (LEAST COST) BRANCH AND BOUND SEARCH

- In both FIFO and LIFO Branch and Bound the selection rules for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preferences to a node that has a very good chance of getting the search to an answer node quickly.
- In this we will use ranking function or cost function. We generate the children of E-node, among these live nodes; we select a node which has minimum cost. By using ranking function we will calculate the cost of each node.

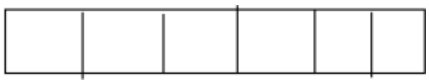


- Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4. By using ranking function we will calculate the cost of 2, 3, 4 nodes is ĉ =2, ĉ =3, ĉ =4 respectively.
- Now we will select a node which has minimum cost i.,e node 2. For node 2, the children are 5, 6. Between 5 and 6 we willselect the node 6 since its cost minimum.
- Generate children of node 6 i.,e 12 and 13. We will select node 12 since its cost (ĉ =1) is minimum. More over 12 is the answer node. So, we terminate search process.

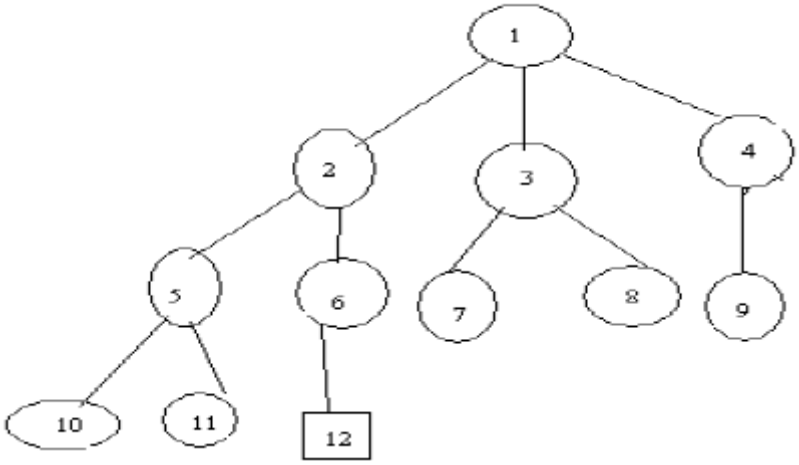
```
listnode = record {
    listnode * next, * parent; float cost;
}

Algorithm LCSearch(t)
// Search t for an answer node.
{
    if *t is an answer node then output *t and return;
    E := t; // E-node.
    Initialize the list of live nodes to be empty;
    repeat
    {
        for each child x of E do
        {
            if x is an answer node then output the path
                from x to t and return;
            Add(x); // x is a new live node.
            (x → parent) := E; // Pointer for path to root.
        }
        if there are no more live nodes then
        {
            write ("No answer node"); return;
        }
        E := Least();
    } until (false);
}
```

5.8. FIFO BRANCH AND BOUND SEARCH

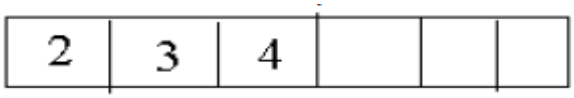
- For this we will use a data structure called Queue. Initially Queue isempty.



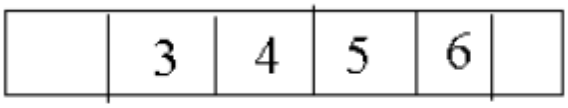
Example:



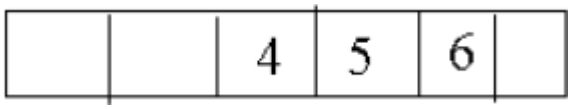
- Assume the node 12 is an answer node (solution)
- In FIFO search, first we will take E-node as a node 1.
- Next we generate the children of node 1. We will place all these live nodes in a queue.



- Now we will delete an element from queue, i.e. node 2, next generate children of node 2 and place in thisqueue.



- Next, delete an element from queue and take it as E-node, generate the children of node 3, 7, 8 are childrenof 3 and these live nodes are killed by bounding functions. So we will not include in the queue.



- Again delete an element an from queue. Take it as E-node, generate the children of 4. Node 9 is generatedand killed by boundary function.

- Next, delete an element from queue.
- Generate children of nodes 5, i.e., nodes 10 and 11 are generated and by boundary function, last node in queue is 6.
- The child of node 6 is 12 and it satisfies the conditions ofthe problem, which is the answer node, so search terminates.

BOUNDING

A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost $c(x)$ associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. (Another method, heuristic search, is discussed in the exercises.) A cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If *upper* is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > upper$ may be killed as all answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > upper$. The starting value for *upper* can be obtained by some heuristic or can be set to ∞. Clearly, so long as the initial value for *upper* is no less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of *upper* can be updated.

Let us see how these ideas can be used to arrive at branch-and-bound algorithms for optimization problems. In this section we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function. We need to be able to formulate the search for an optimal solution as a search

for a least-cost answer node in a state space tree. To do this, it is necessary to define the cost function $c(\cdot)$ such that $c(x)$ is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for $c(\cdot)$. For nodes representing feasible solutions, $c(x)$ is the value of the objective function for that feasible solution. For nodes representing infeasible solutions, $c(x) = \infty$. For nodes representing partial solutions, $c(x)$ is the cost of the minimum-cost node in the subtree with root x. Since $c(x)$ is in general as hard to compute as the original optimization problem is to solve, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x. In general then, the $\hat{c}(\cdot)$ function used in a branch-and-bound solution to optimization functions will estimate the objective function value and not the computational difficulty of reaching an answer node. In addition, to be consistent with the terminology used in connection with the 15-puzzle, any node representing a feasible solution (a solution node) will be an answer node. However, only minimum-cost answer nodes will correspond to an optimal solution. Thus, answer nodes and solution nodes are indistinguishable.