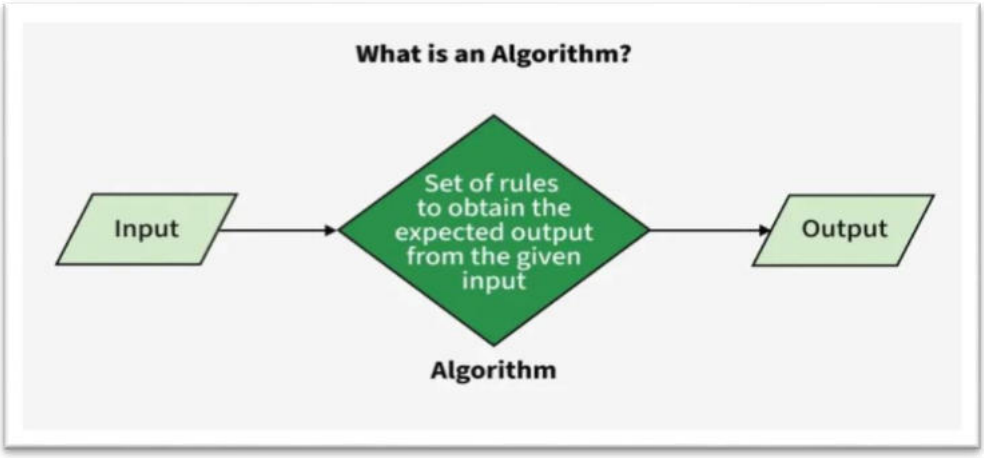
UNIT – I

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations.
AVL Trees – Creation, Insertion, Deletion operations and Applications
B-Trees – Creation, Insertion, Deletion operations and Applications.

Introduction to Algorithms:

The word **Algorithm** means "A set of finite rules or instructions to be followed in calculations or other problem-solving operations" Or "A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations".

Therefore Algorithm refers to a sequence of finite steps to solve a particular problem.



Use of the Algorithms:

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

1. **Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
2. **Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
3. **Operations Research**: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
4. **Artificial Intelligence:** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
5. **Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

These are just a few examples of the many applications of algorithms. The use of algorithms is continually expanding as new technologies and fields emerge, making it a vital component of modern society.

Algorithms can be simple and complex depending on what you want to achieve.

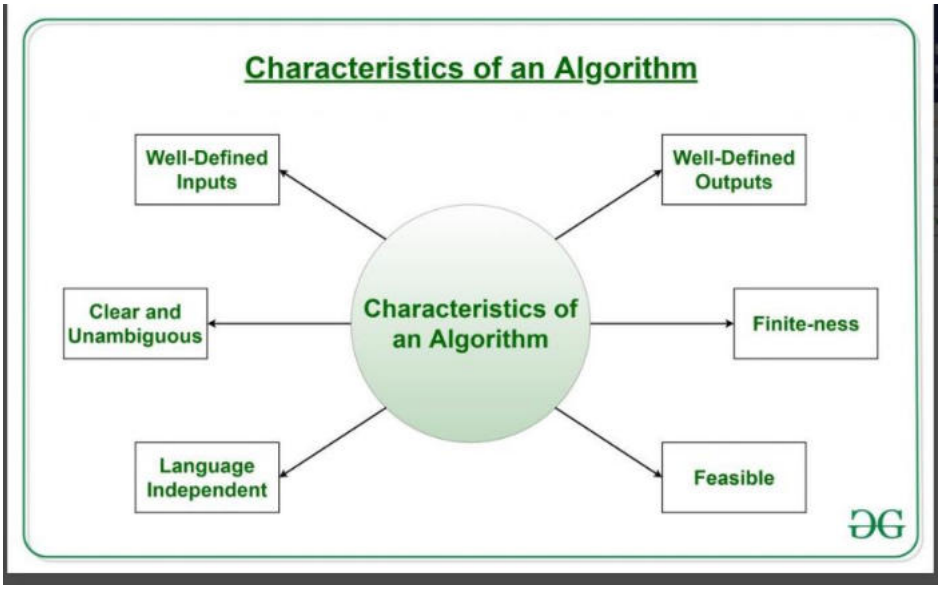
It can be understood by taking the example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and executes them one by one, in the given sequence. The result thus obtained is the new dish is cooked perfectly. Every time you use your phone, computer, laptop, or calculator you are using Algorithms. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

What is the need for algorithms?

1. Algorithms are necessary for solving complex problems efficiently and effectively.
2. They help to automate processes and make them more reliable, faster, and easier to perform.
3. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
4. 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.

What are the 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**).

How to Design an Algorithm?

To write an algorithm, the following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm i.e. clear problem definition.
2. The **constraints** of the problem must be considered while solving the problem.
3. The **input** to be taken to solve the problem.

4. The **output** is to be expected when the problem is solved.
5. The **solution** to this problem is within the given constraints.

Then the algorithm is written with the help of the above parameters such that it solves the problem.

Example: Consider the example to add three numbers and print the sum.

Step 1: Fulfilling the pre-requisites

As discussed above, to write an algorithm, its prerequisites must be fulfilled.

1. **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
2. **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
3. **The input to be taken to solve the problem:** The three numbers to be added.
4. **The output to be expected when the problem is solved:** The sum of the three numbers taken as the input i.e. a single integer value.
5. **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of the '+' operator, or bit-wise, or any other method.

Step 2: Designing the algorithm

Now let's design the algorithm with the help of the above pre-requisites:

Algorithm to add 3 numbers and print their sum:

1. START
2. Declare 3 integer variables num1, num2, and num3.
3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of the variable sum
7. END

Step 3: Testing the algorithm by implementing it.

Performance Analysis:

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to 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 that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm 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**)

Space Complexity:

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.

3. To store variable values.
4. And for few other things like funcion 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 used for **return value**.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any 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 of memory 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.

Time Complexity:

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer 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 time complexity 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 model machine...

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.

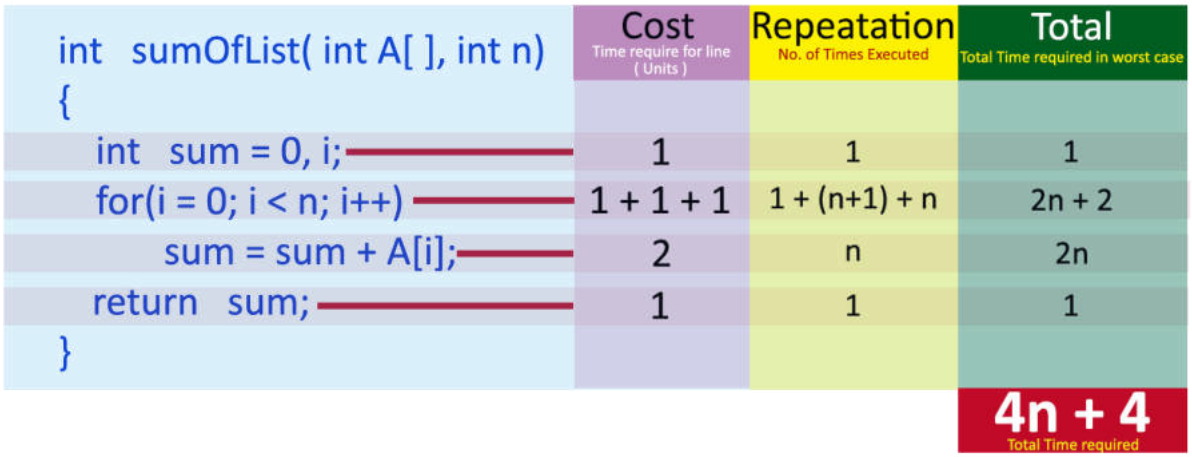
If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

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.

Asymptotic Notations:

What is Asymptotic Notation?

Whenever we want 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 has least 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 (**$5n^2$** 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 terms to 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, we use THREE types of Asymptotic Notations and those are as follows...

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

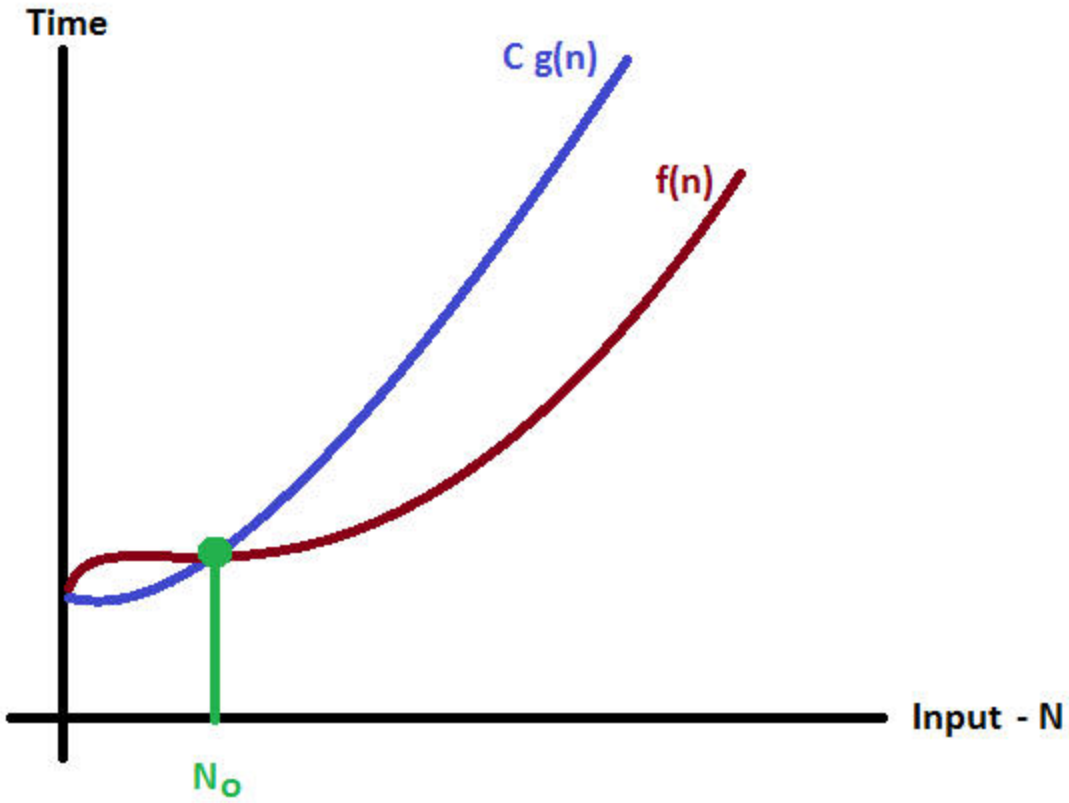
Big - Oh Notation (O)

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 time required 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)

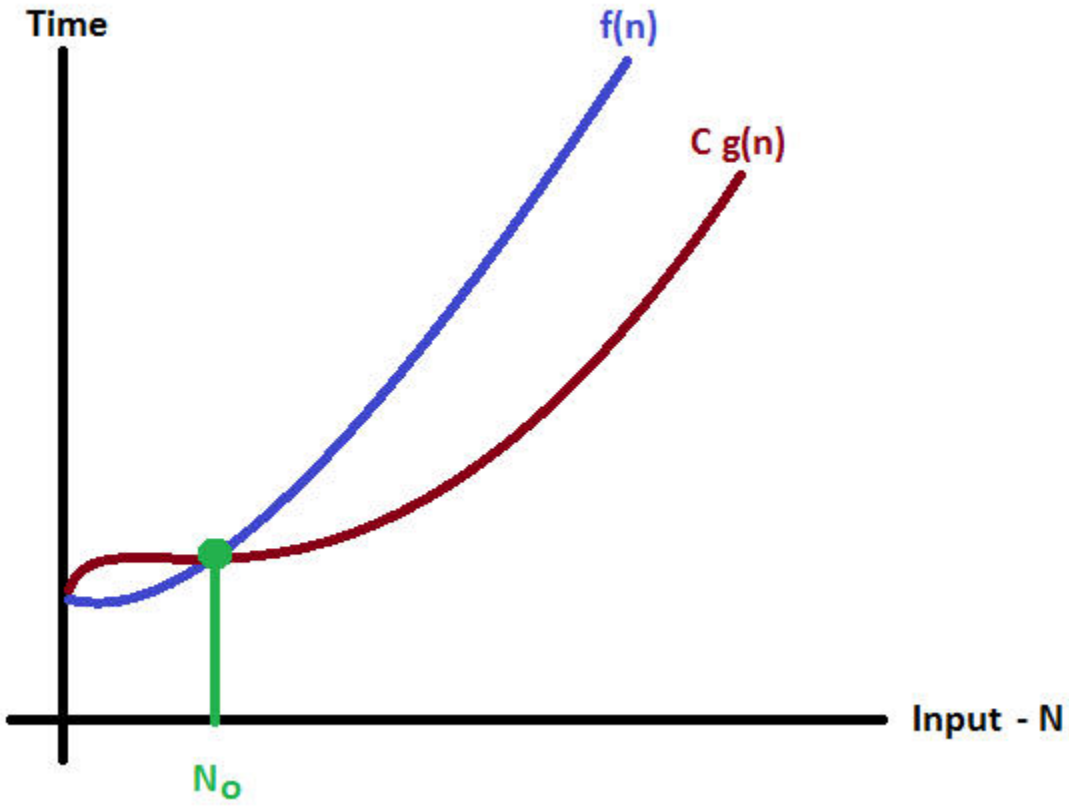
Big - Omege Notation (Ω)

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 time required 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 the algorithm'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)

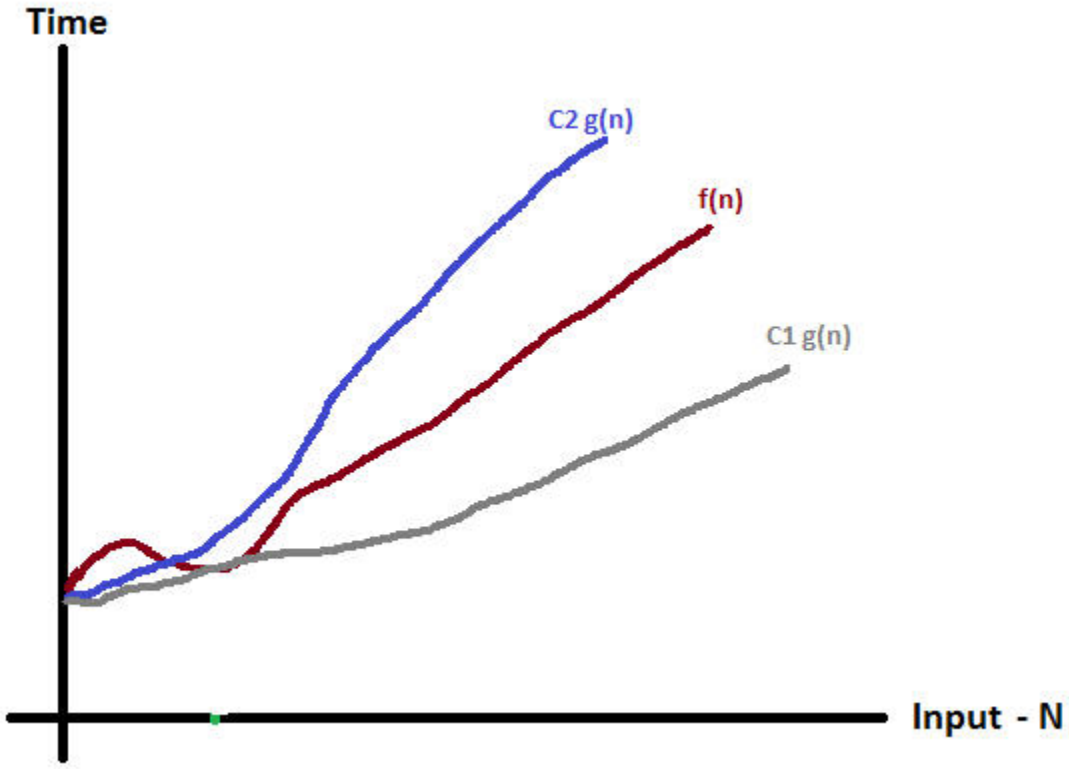
Big - Theta Notation (Θ)

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 time required 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 than f(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)

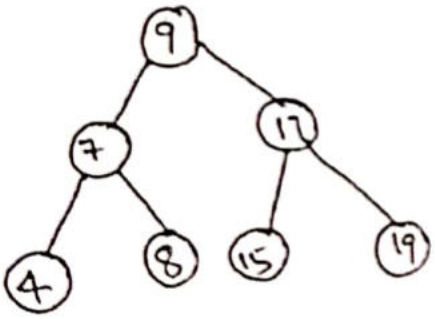
AVL Trees :-

The binary search tree provides a convenient structure for storing and searching data collections.

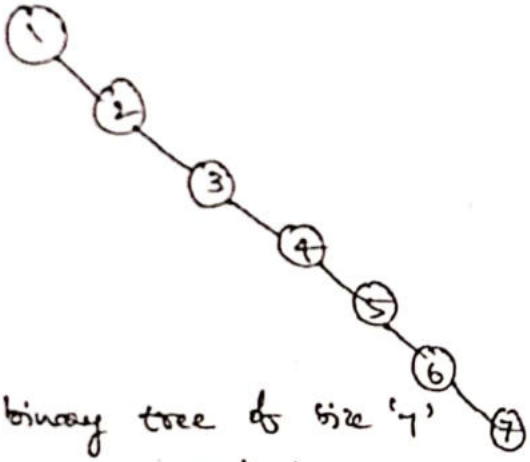
→ The efficiency of the search, insertion, and deletion operations depend on the height of the tree.

→ In the best case, a binary tree of size 'n' has a height of $\log n$.

→ In the worst case, there is one node per level, resulting in a height of 'n'.



A binary tree of size '7' with height of '3'.



A binary tree of size '7' with height of '7'.

The worst case complexity of binary search tree, constructed with height 'n'.

To overcome the worst case height of the Binary search Tree, the height of the binary search tree balanced. To do this, Height balanced trees are constructed instead of Binary search trees.

Examples of Height balanced trees.

(1) AVL Trees
(2) Splay Trees
(3) Red-Black Trees.

AVL Trees:

AVL trees are height balanced binary search trees.

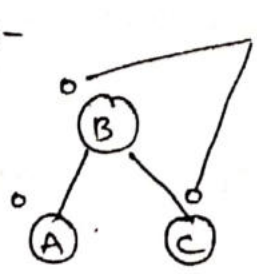
→ AVL stands for Adelson, Velski, Landis.

→ AVL tree checks the height of the left and the right subtrees is not more than 1 at each node of the tree.

→ The difference of the height of the left subtree and right subtree called as "Balance factor" which is not more than 1.

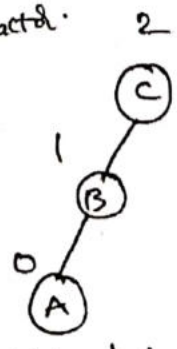
> Balance Factor (BF) = Height of left tree − Height of right subtree

Ex:-



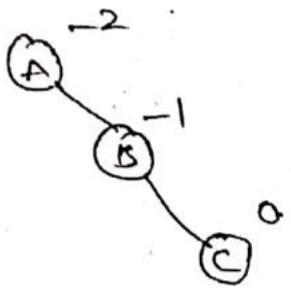
Balanced Tree

Balance Factor(B) = 0-0 = 0
Balance Factor(A) = 0-0 = 0
Balance Factor(C) = 0-0 = 0



Not balanced

Balance Factor(C) = 2-0 = 0
Balance Factor(B) = 1-0 = 1
Balance Factor(A) = 0-0 = 0



Not balanced

Balance Factor(A) = 0-2 = -2

AVL Rotations:-

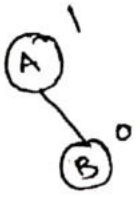
→ To balance height of the tree itself, an AVL tree perform 4 kinds of rotations.

(i) Left-Left rotation (LL Rotation)
(ii) Right-Right rotation (RR Rotation)
(iii) Left-Right rotation (LR Rotation)
(iv) Right-Left rotation (RL Rotation)

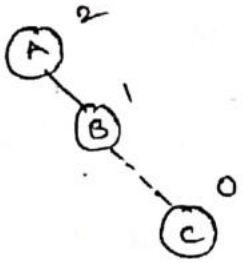
For program refer page no: 76

(i) Left-Left rotation:-

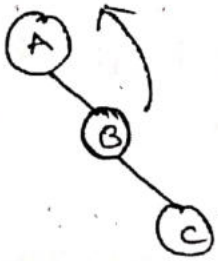
"If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree" then we perform a single left rotation.



Tree balanced

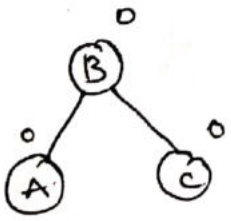


Tree Unbalanced when Inserting 'C'.



perform Left Rotation

⇒

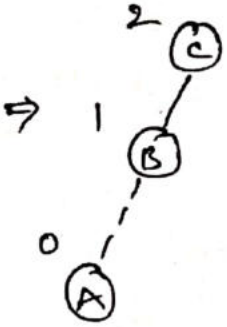


(ii) Right-Right Rotation:

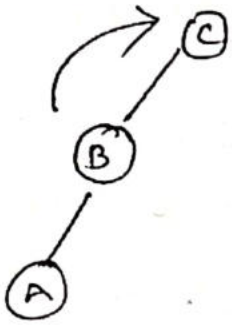
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

Scanned with CamScanner

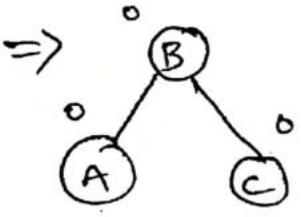
Tree Balanced



Tree un balanced when Inserting 'A'.



perform Right Rotation.



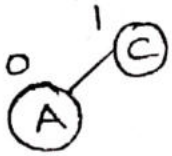
Balanced Tree.

(iii) Left - Right Rotation :-

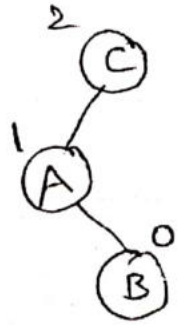
A Left-Right Rotation is a combination of left rotation followed by right rotation.

⇒ This LR rotation performed when a node has been Inserted into the right subtree of left subtree.

Ex:-



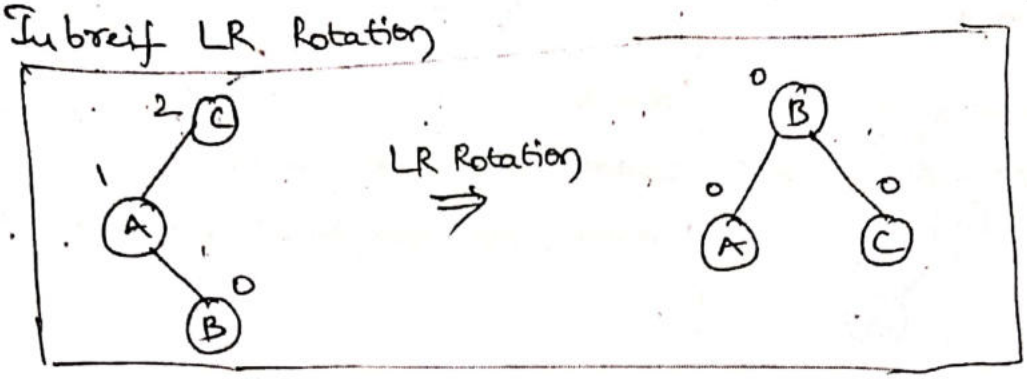
Tree balanced



Tree unbalanced when Inserted 'B'.

⇒ This case balanced through LR Rotation.

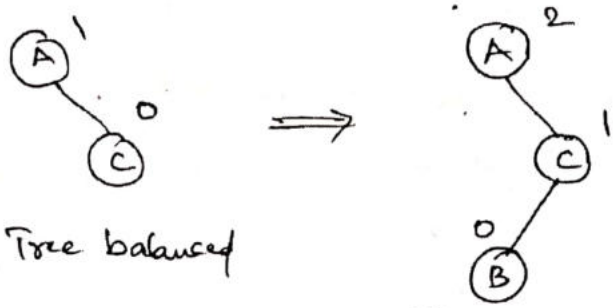
In brief LR Rotation



(iv) Right - Left Rotation:

A Right-left Rotation is a combination of right rotation followed by left rotation.

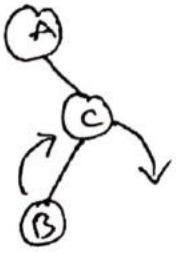
Ex:-

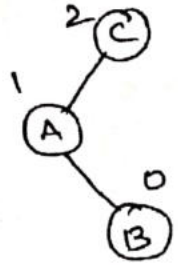
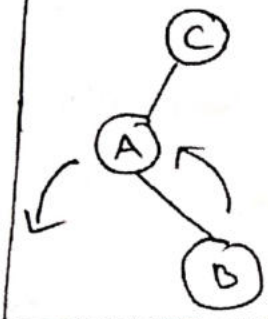
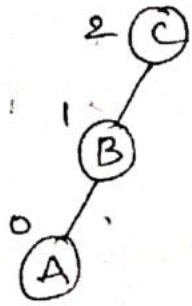
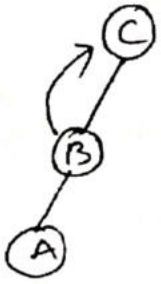
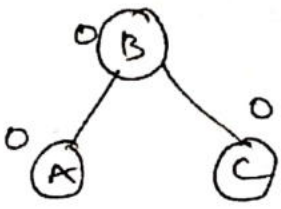


Tree balanced

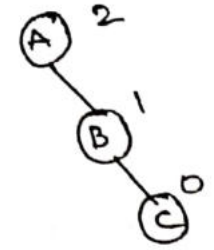
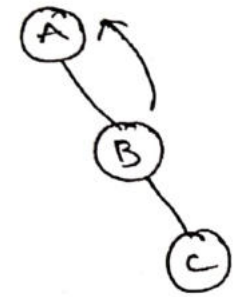
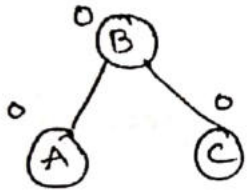
This case ⇒ balanced through RL Rotation.

Tree unbalanced when inserted 'B'.

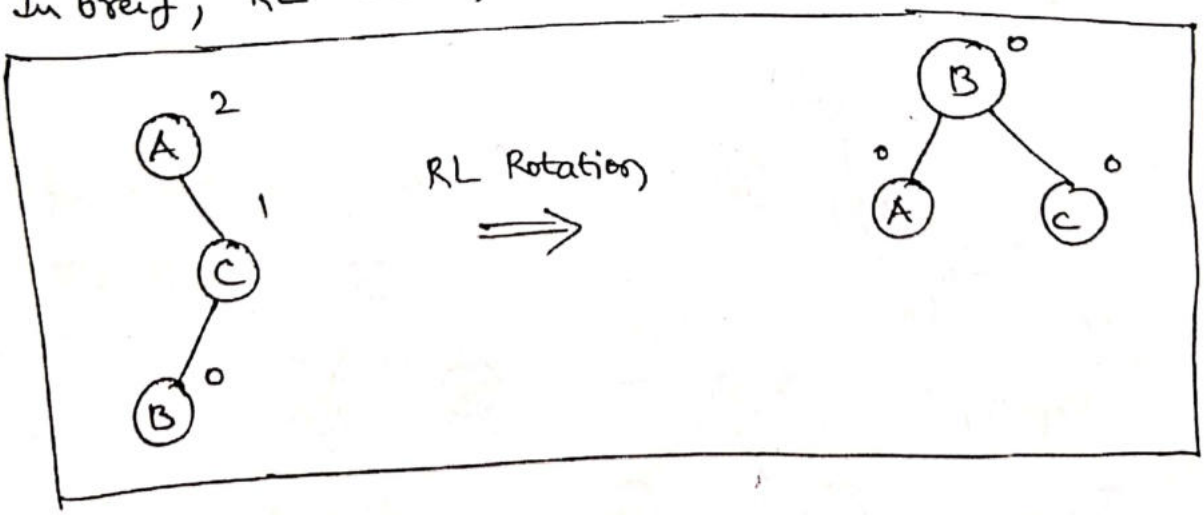
State	Action
	A node has been inserted into the left subtree of the right subtree. This makes unbalanced at node 'A'.
	perform right rotation at node 'C'.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes an unbalanced tree at node 'C'.
	Perform Left rotation on the left sub tree of C.
	Node C is still unbalanced, because of the left subtree of left subtree.
	Perform Right rotation, making B as the new root node of this subtree
	The tree is now balanced

Scanned with CamScanner

State	Action
	Node 'A' is still unbalanced because of the right subtree of ~~the~~ its right subtree and requires a left rotation
	perform Left rotation
	Now the tree is balanced.

In brief, RL rotation



Construct AVL tree for the following sequence of elements: (61)

50, 20, 60, 10, 8, 15, 32, 46, 11, 48.

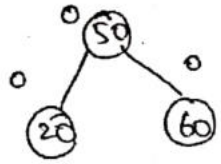
Step 1) Insert 50

Tree is balanced

Step 2) Insert 20

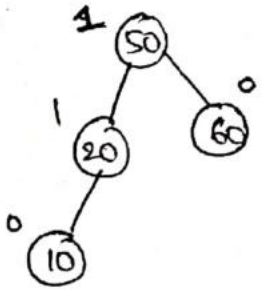
Tree is balanced.

Step 3) Insert 60



Tree is balanced

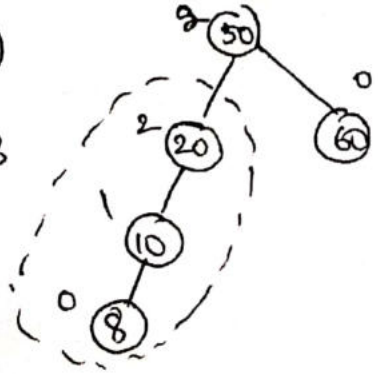
Step 4) Insert 10



Tree is balanced

Step 5)

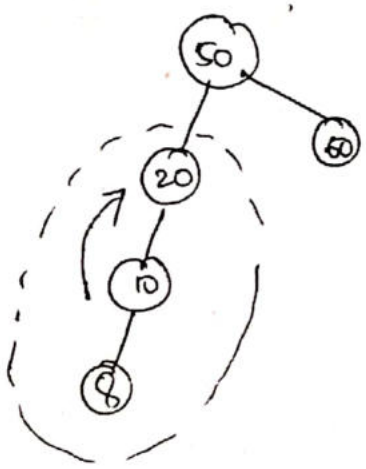
Insert 8



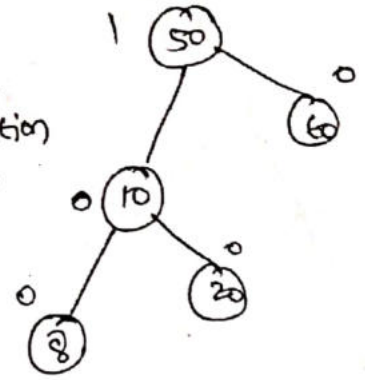
Tree unbalanced at node 20.

Step 5) Continuation

Perform Right Rotation

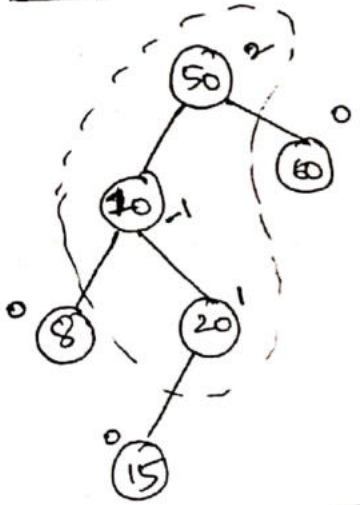


RR Rotation ⇒



Now the tree is balanced.

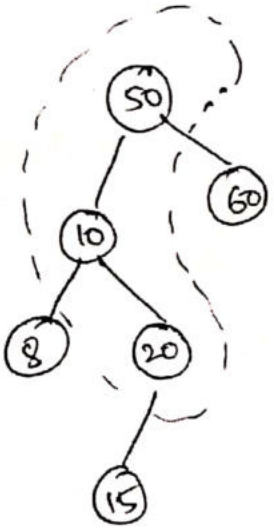
Step 6) Insert 15



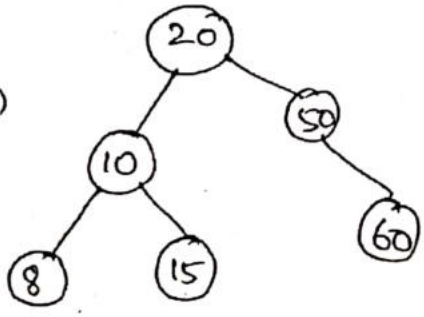
Tree unbalanced at node '50'.

Perform LR Rotation.

Step 7)

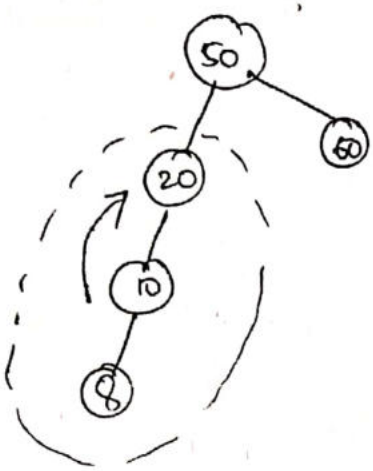


LR Rotation ⇒

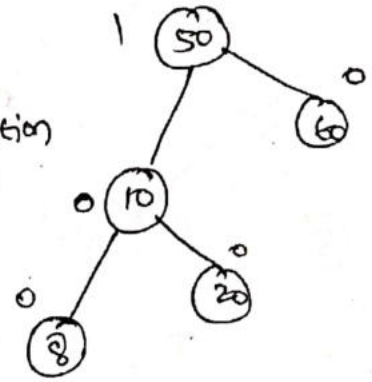


Step 5) continuation

Perform Right Rotation

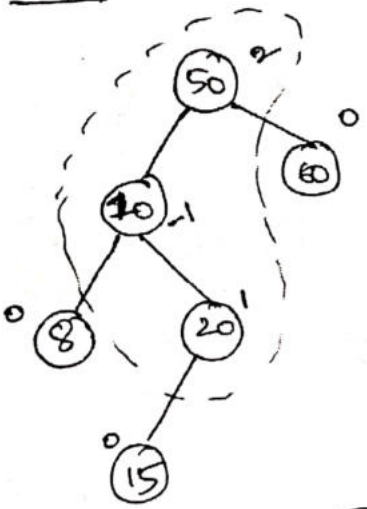


RR Rotation ⇒



Now the tree is balanced.

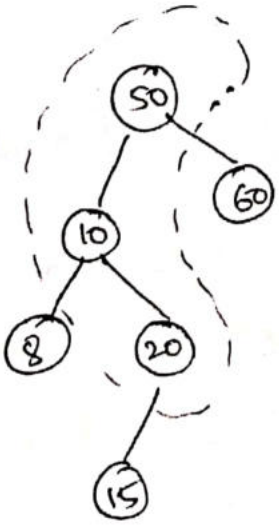
Step 6) Insert 15



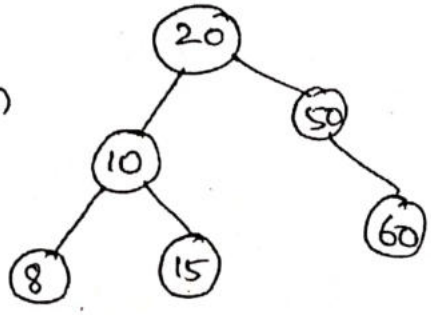
Tree unbalanced at node '50'.

Perform LR Rotation.

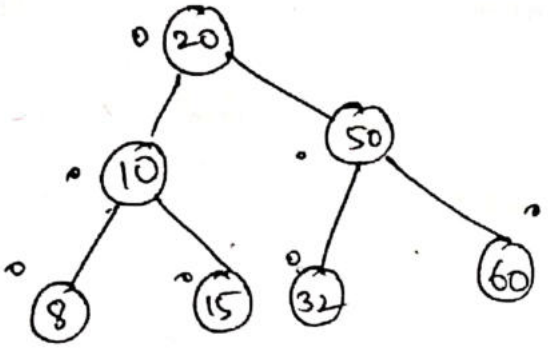
Step 7)



LR Rotation ⟹

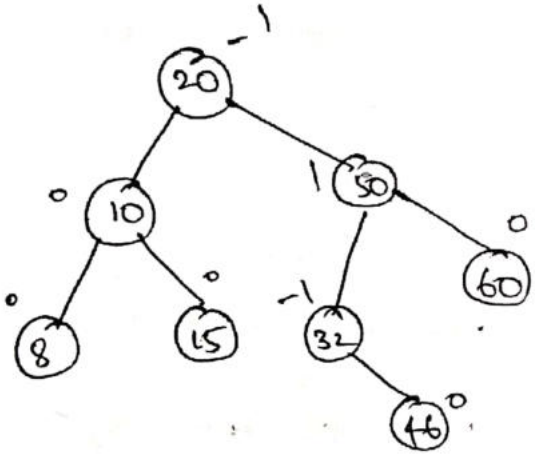


Step 7) Insert 32



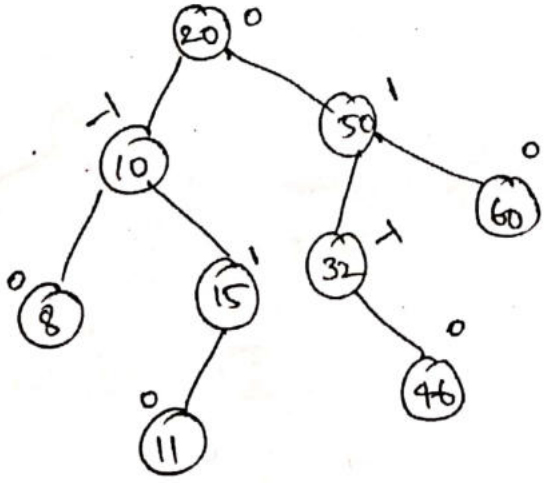
Tree is balanced

Step 8) Insert 46



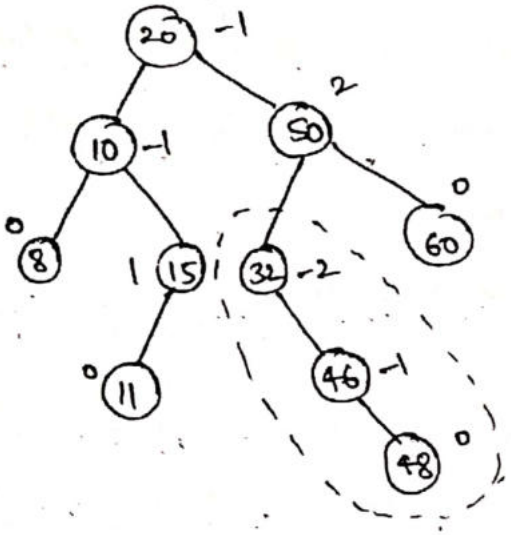
Tree is balanced.

Step 9) Insert 11



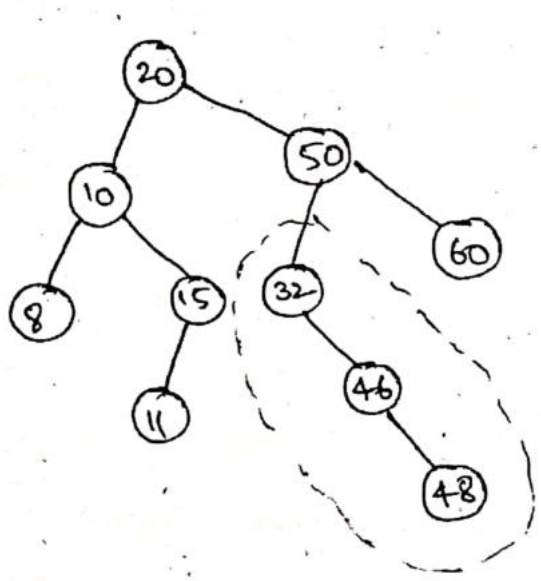
Tree is balanced.

Step 10) Insert 48

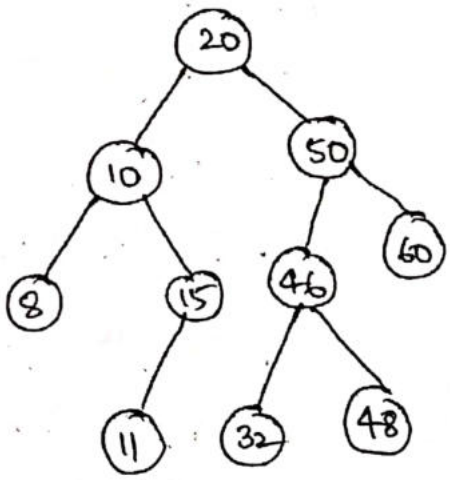


Tree is unbalanced

Perform LL Rotation



LL Rotation ⟹



Example problems :-

Construct AVL tree for the following elements.

(i) 14, 17, 11, 7, 53, 4, 13.

(ii) c, a, d, f, e, t, j, K, S, i.

AVL Tree deletion operation:

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced.

⇒ There are two different rotations possible when deleting a node from AVL tree.

(i) If the node which is to be deleted is present in the left subtree of the critical node, then L rotation needs to be applied.

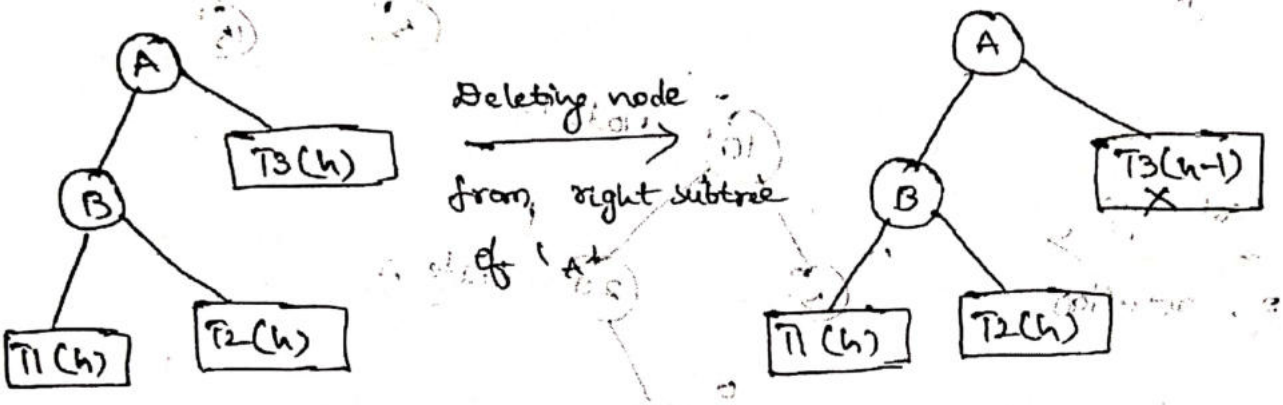
(ii) If the node which is to be deleted is present in the right subtree of the critical node, then R rotation will be applied.

Let us consider, A is the critical node and B is the root node of its left subtree. If the node 'X', present in the right subtree of A, is to be deleted, then there can be three different situations.

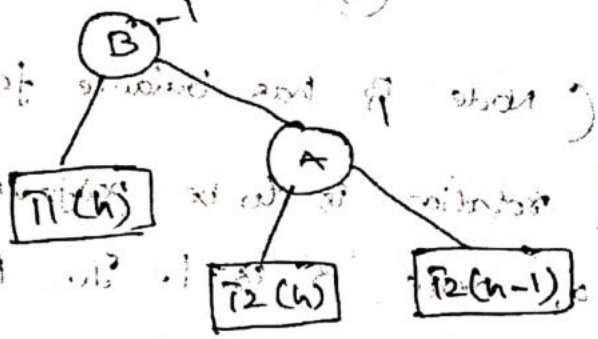
(i) RO rotation (Node B has balance factor 0):

If the node B has '0' balance factor and the balance factor of node 'A' disturbed upon deleting the node 'X', then the tree will be rebalanced by rotating tree using 'RO' rotation.

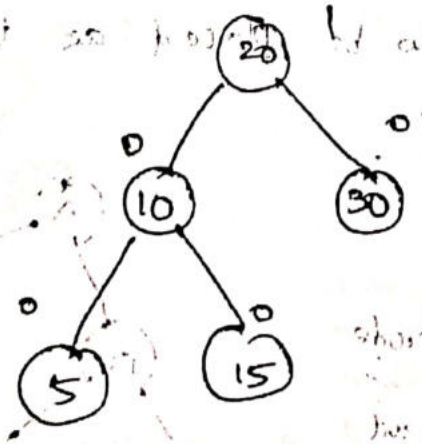
⇒ The critical node 'A' is moved to its right, and the node 'B' becomes the root of the tree with T1 as its left subtree. The subtrees T2 and T3 becomes the left and right subtree of the node 'A'



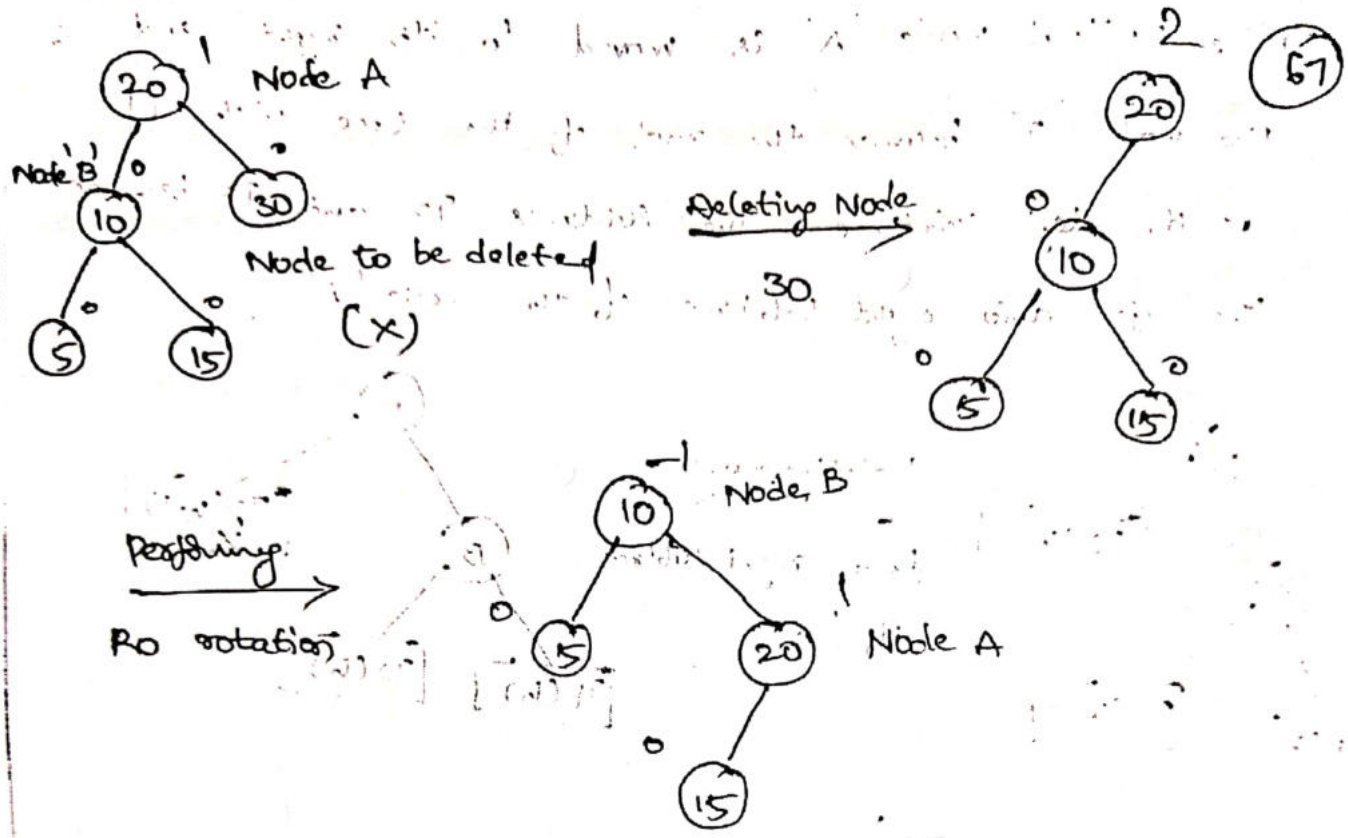
Perform RO rotation:



Example:- Deleting the node 30 from the AVL tree

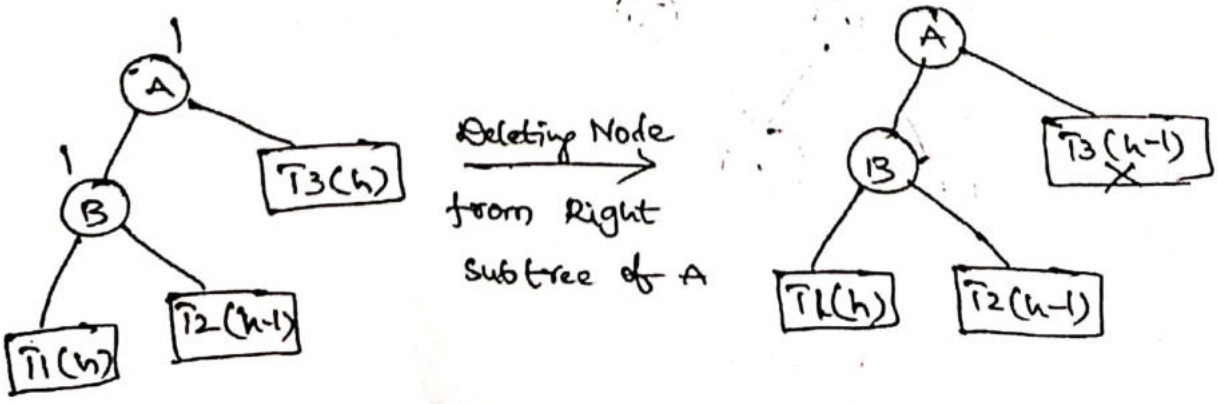


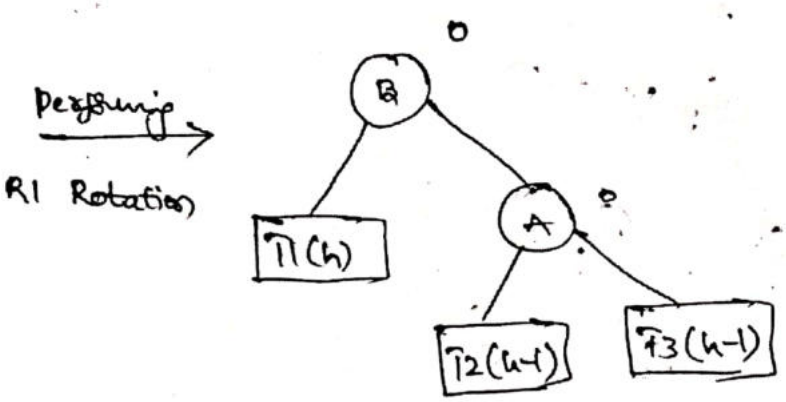
Scanned with CamScanner



(i) RI Rotation :- (Node B has balance factor 1)

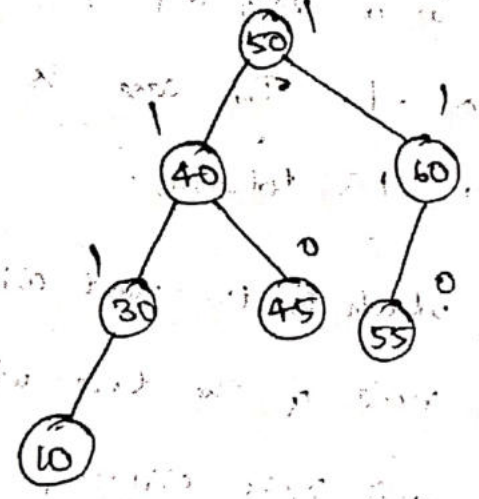
RI rotation is to be performed if the balance factor of Node 'B' is 1. In RI rotation, the vertical node 'A' is moved to its right having subtrees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub tree of the node 'B'.

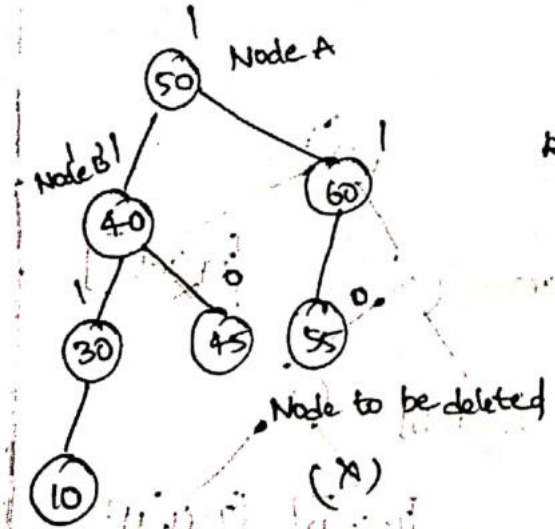


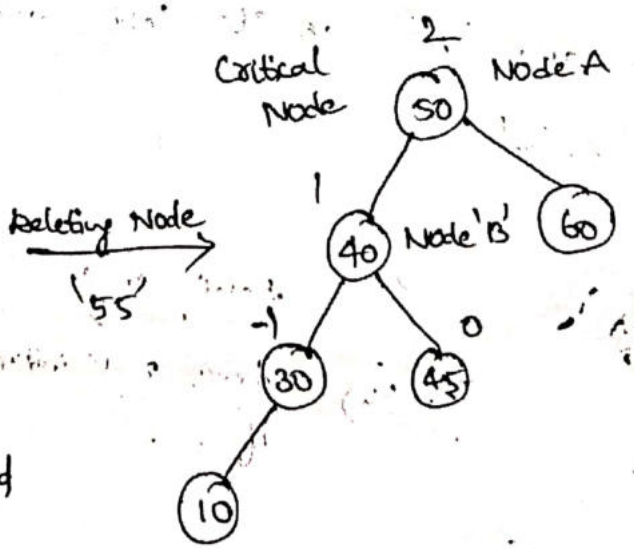


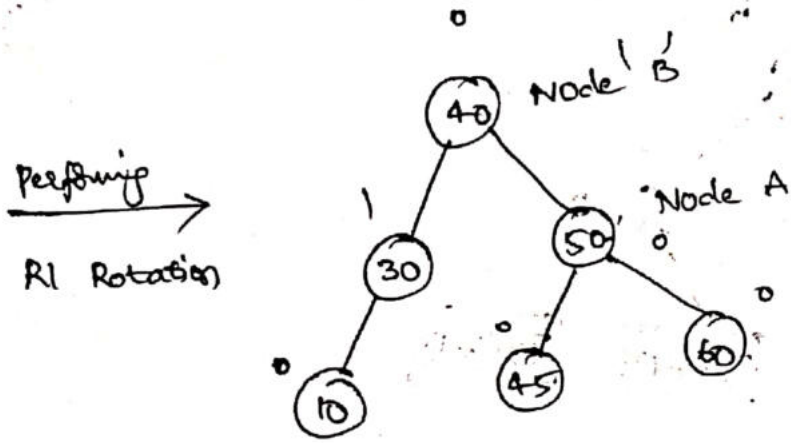
Example:

Delete Node 55 from the AVL tree.





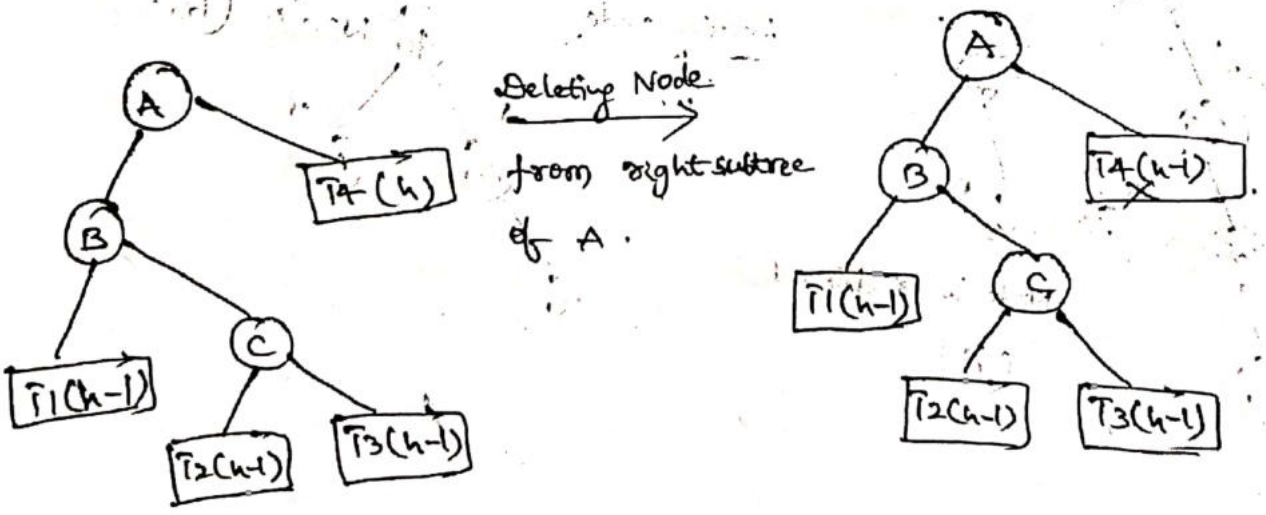


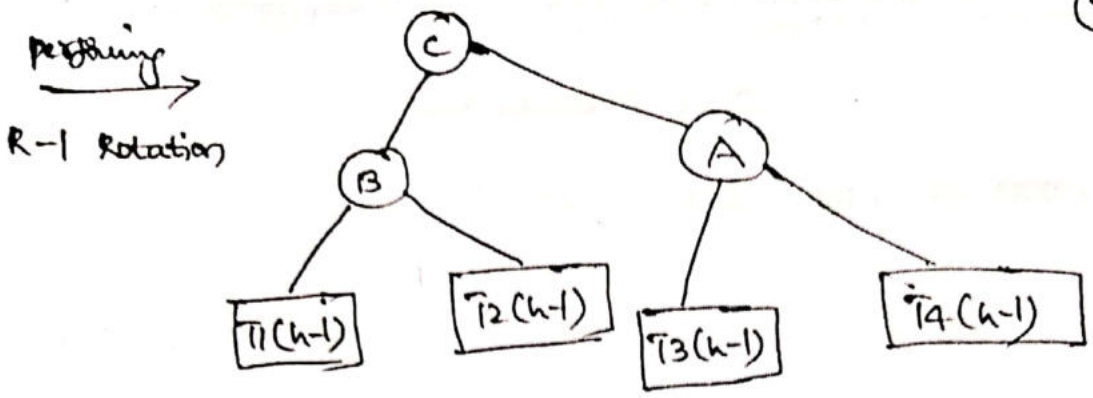


R-1 Rotation (Node B has balance factor -1)

R-1 rotation is to be performed if the node 'B' has balance factor -1. This case is treated in the same way as LR Rotation.

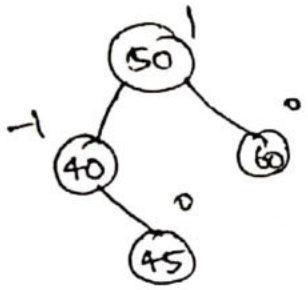
In this case, the node 'C', which is the right child of node B, becomes the root node of the tree with 'B' and 'A' as its left and right children respectively.

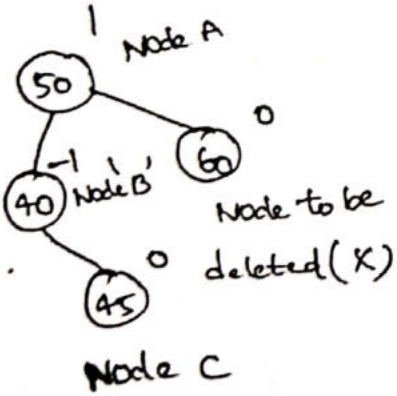


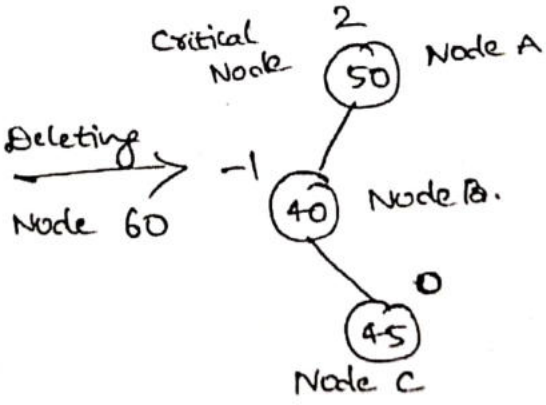


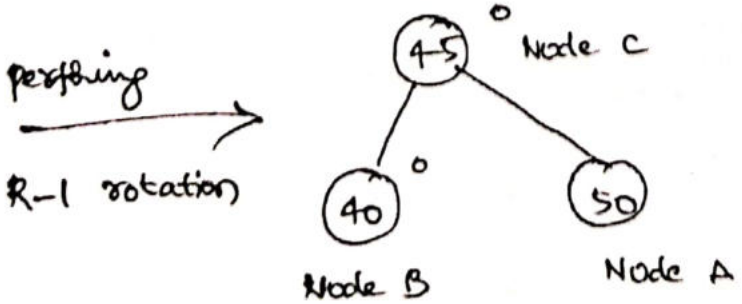
Example:

Delete the node '60' from the AVL tree:









Applications of AVL Tree:

1. AVL Tree is used as a first example self balancing BST in teaching DSA as it is easier to understand and implement compared to Red Black
2. Applications, where insertions and deletions are less common but frequent data lookups along with other operations of BST like sorted traversal, floor, ceil, min and max.
3. Red Black tree is more commonly implemented in language libraries like map in C++, set in C++, TreeMap in Java and TreeSet in Java.
4. AVL Trees can be used in a real time environment where predictable and consistent performance is required.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves and therefore provides time complexity as **O(log n)** for **search**, **insert** and **delete**.
2. As it is a balanced BST, so items can be traversed in sorted order.
3. Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.
4. AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

Disadvantages of AVL Tree:

1. It is difficult to implement compared to normal BST.
2. Less used compared to Red-Black trees. Due to its rather strict balance.
3. AVL trees provide complicated insertion and removal operations as more rotations are performed.

B Trees:

A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. It is commonly used in databases and file systems.

B-trees are always balanced, meaning that all leaf nodes are at the same depth. This ensures that the time complexity for search, insert, delete, and traverse operations remains efficient.

Properties of B Tree:

The following are some important properties of a B Tree:

1. Every node has at most m children, where m is the order of the B Tree.
2. A node having K children consists of K-1 keys.
3. Every non-leaf node, excluding the root node, must have at least [m/2] child nodes.
4. The root node must have at least two children if it is not the leaf node.

5. Unlike the other trees, the height of a B Tree increases upwards toward the root node, and the insertion happens at the leaf node.
6. The Time Complexity of all the operations of a B Tree is O(log n), where 'n' is the number of data elements present in the B Tree.

Insertion in B-Trees:

1. **Find the Leaf Node:**
    - Traverse the tree to find the appropriate leaf node for the new key.
2. **Insert the Key:**
    - Add the key in sorted order within the leaf node.
3. **Handle Overflow by Splitting:**
    - If the leaf node is full, split it into two nodes, and promote the middle key to the parent.
4. **Propagate Splits Upward:**
    - If necessary, continue splitting up the tree, which may result in creating a new root.

Let us understand the steps mentioned above with the illustrations shown below.

Suppose that the following are some data elements that need to be inserted in a B Tree: **7, 8, 9, 10, 11, 16, 21, and 18**.

1. Since the maximum degree of a node in the tree is **3**; therefore, the maximum number of keys per node will be **3 - 1 = 2**.

2. We will start by inserting data element **7** in the empty tree.

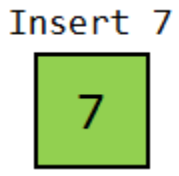


Figure 2.1: Inserting 7

3. We will insert the next data element, i.e., **8**, into the tree. Since **8** is greater than **7**, it will be inserted to the right of **7** in the same node.

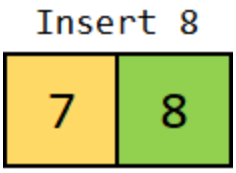


Figure 2.2: Inserting 8

4. Similarly, we will insert another data element, **9**, into the tree on the same to the right of **8**. However, since the maximum number of keys per node can only be **2**, the node will split, pushing the median key **8** upward, making **7** the key of the left child node and **9** the key of the right child node.

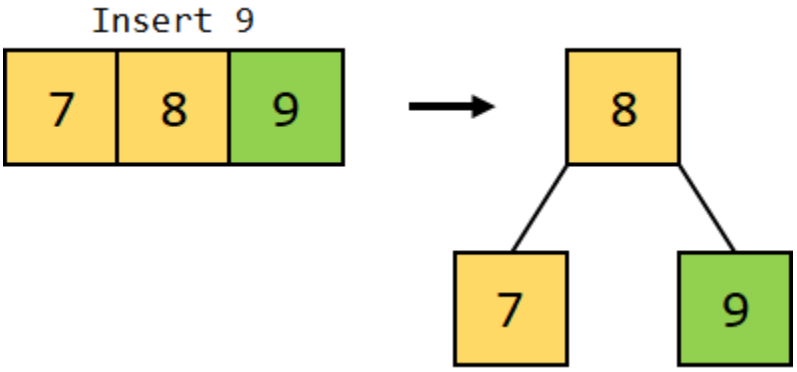


Figure 2.3: Inserting 9

5. We will insert the next data element, i.e., **10**, into the tree. Since **10** is greater than **9**, it will be inserted as a key on the right of the node containing **9** as a key.

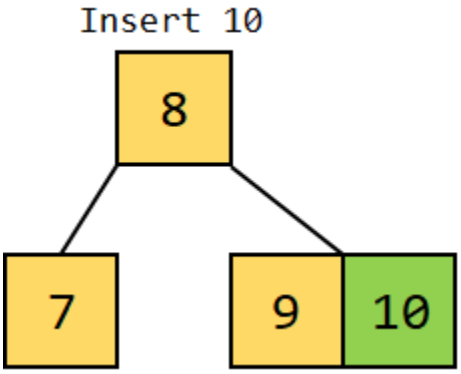


Figure 2.4: Inserting 10

6. We will now insert another data element, **11**, into the tree. Since **11** is greater than **10**, it should be inserted to the right of **10**. However, as we know, the maximum number of keys per node cannot be more than **2**; therefore, **10** being the median, will be pushed to the root node right to **8**, splitting **9** and **11** into two separate nodes.

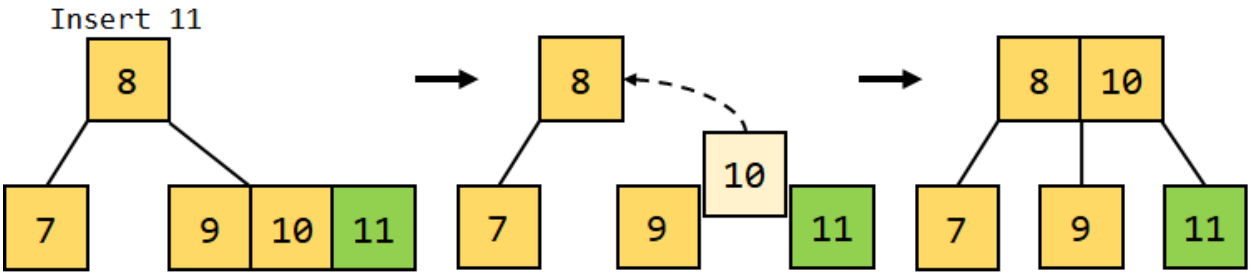


Figure 2.5: Inserting 11

7. We will now insert data element **16** into the tree. Since **16** is greater than **11**, it will be inserted as a key on the right of the node consisting of **11** as a key.

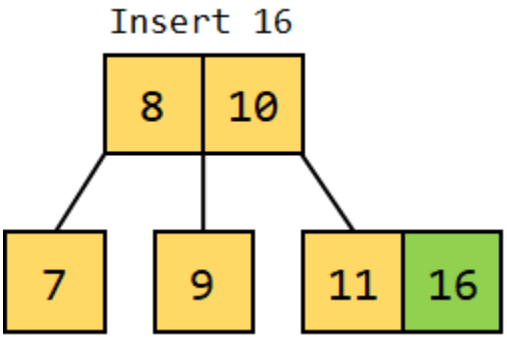


Figure 2.6: Inserting 16

8. The next data element that we will insert into the tree is **21**. Element **21** should be inserted to the right of **16**; however, it will exceed the maximum number of keys per node limit. Therefore, a split will occur, pushing the median key **16** upward and splitting the left and right keys into separate nodes. But this will again violate the maximum number of keys per node limit; therefore, a split will once again push the median key **10** upward a root node and make **8** and **11** its children.

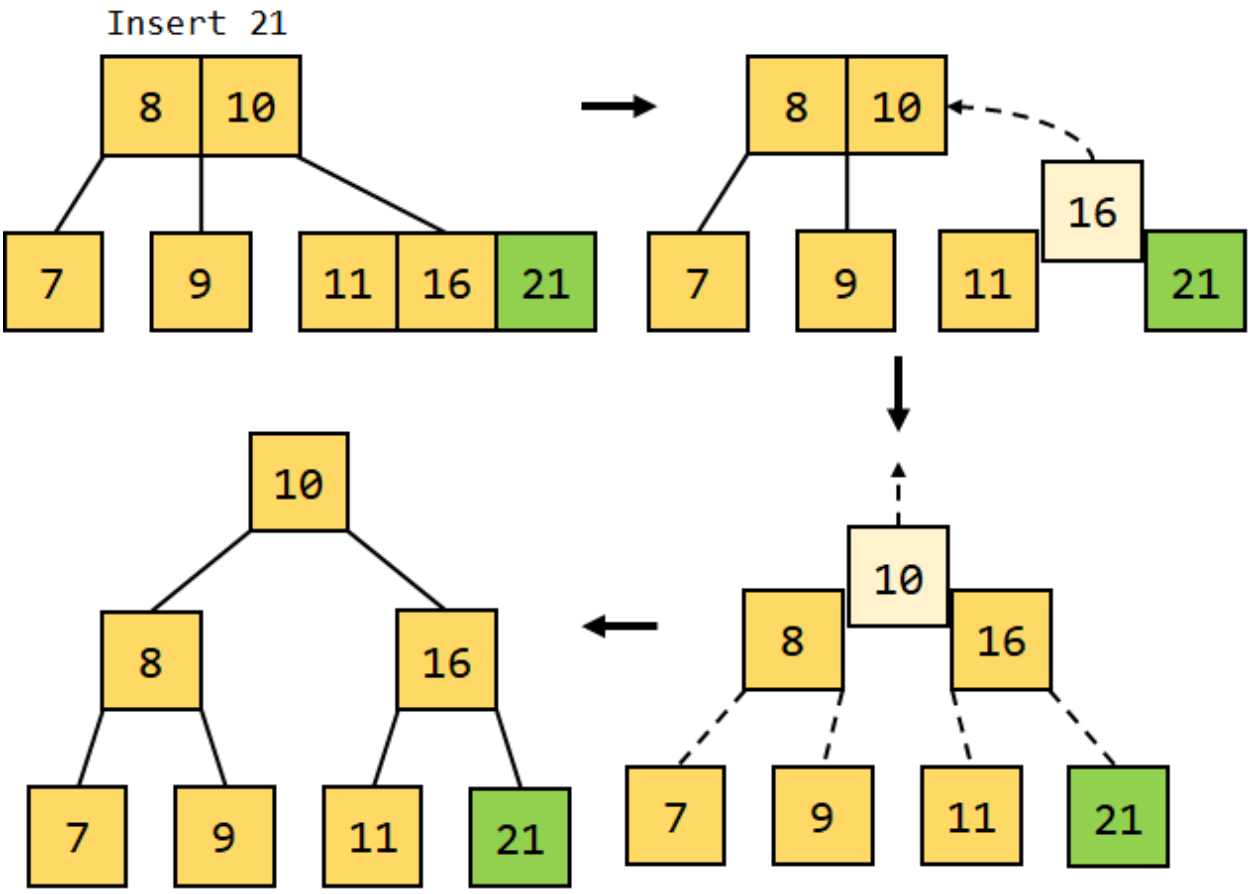


Figure 2.7: Inserting 21

9. At last, we will insert data element **18** into the tree. Since **18** is greater than **16** but less than **21**, it will be inserted as the left key in the node consisting of **21**.

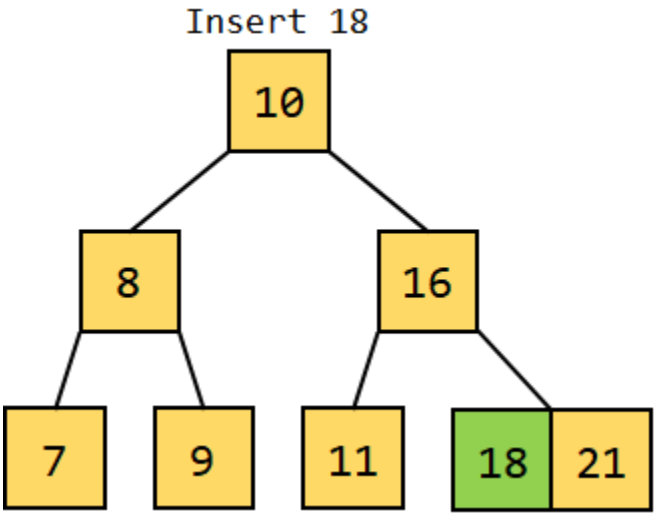


Figure 2.8: Inserting 18

10. Hence the resulted B Tree will be as shown below:

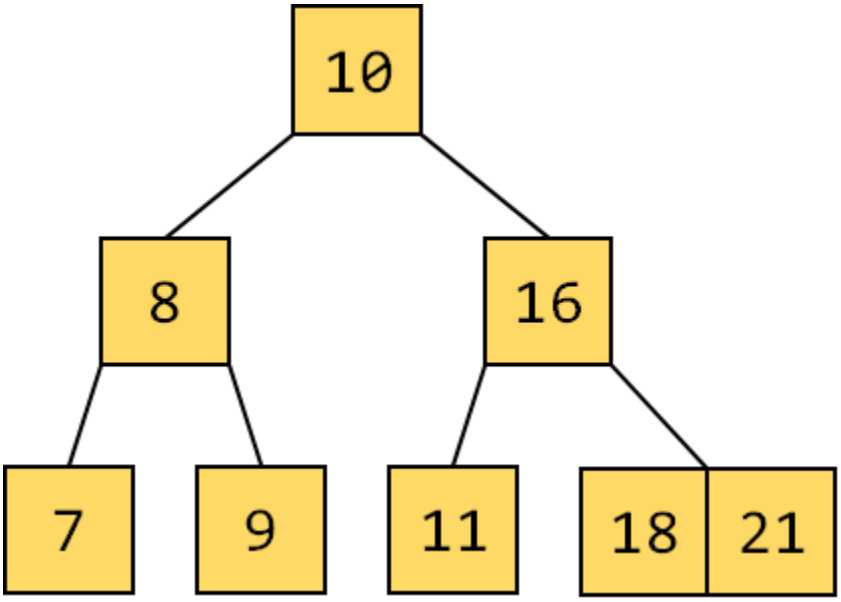


Figure 2.9: The Resulted B Tree

Deletion from a B-tree

Deleting an element on a B-tree consists of three main events: **searching the node where the key to be deleted exists**, deleting the key and balancing the tree if required.

While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.

The terms to be understood before studying deletion operation are:

1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

Before going through the steps below, one must know these facts about a B tree of degree **m**.

1. A node can have a maximum of m children. (i.e. 3)
2. A node can contain a maximum of m - 1 keys. (i.e. 2)
3. A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)
4. A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)

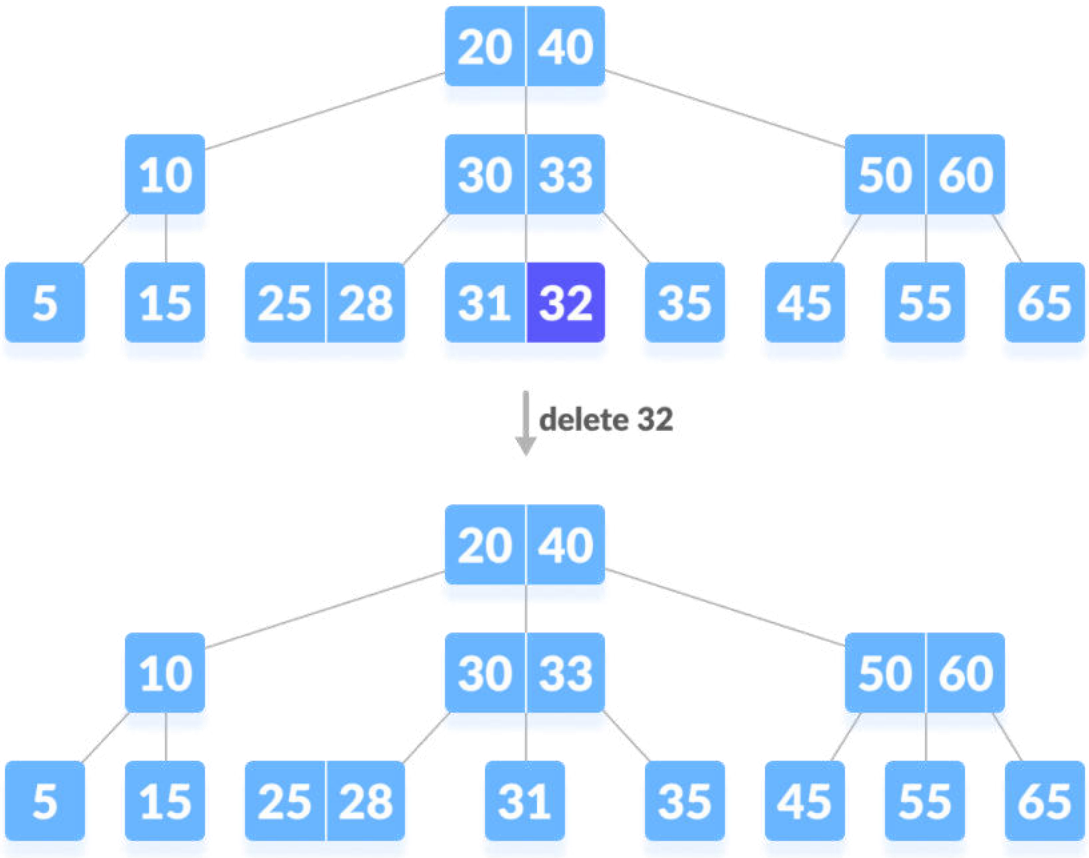
There are three main cases for deletion operation in a B tree.

Case I

The key to be deleted lies in the leaf. There are two cases for it.

1. The deletion of the key does not violate the property of the minimum number of keys a node should hold.

   In the tree below, deleting 32 does not violate the above properties.



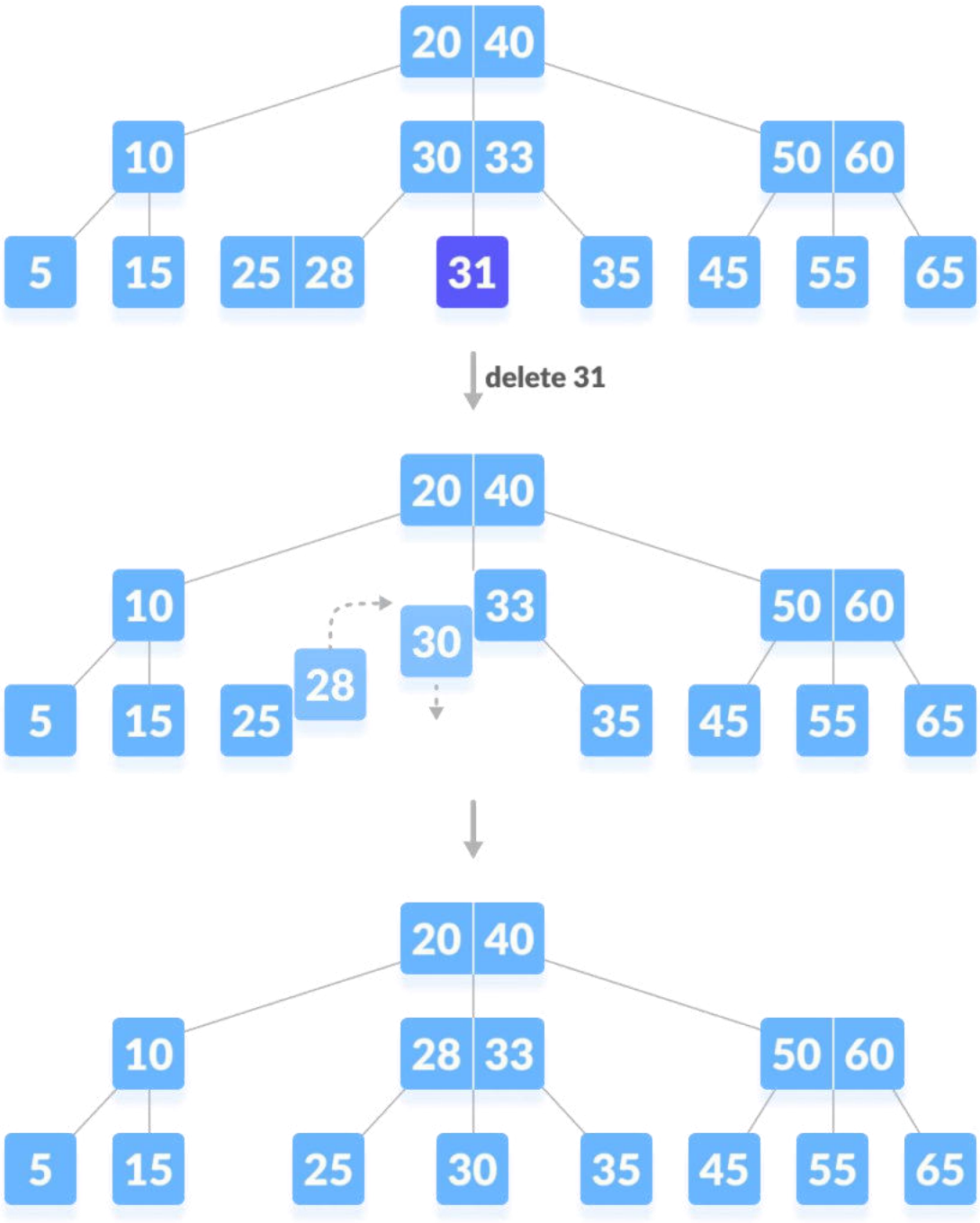
   Deleting a leaf key (32) from B-tree

2. The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

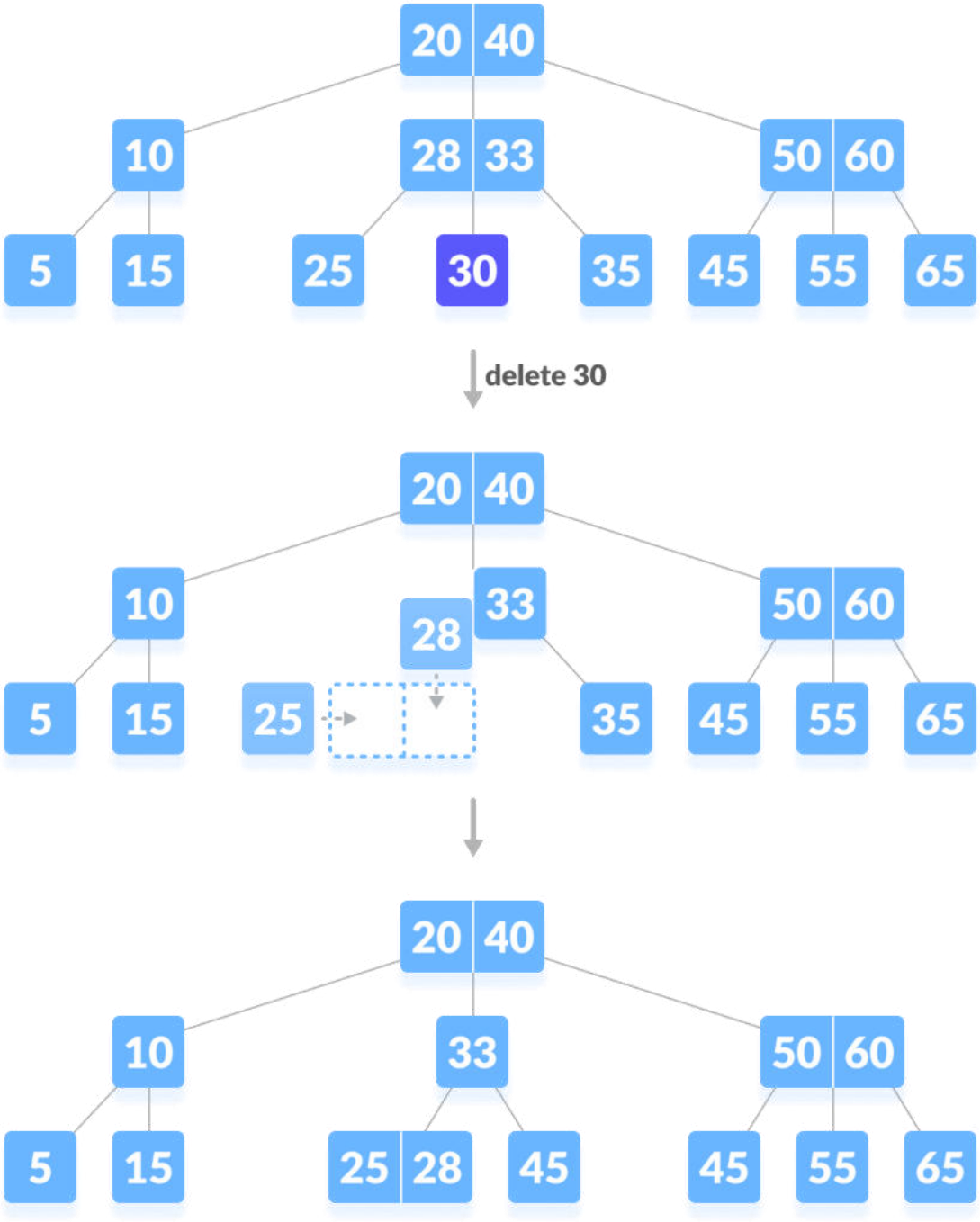
Else, check to borrow from the immediate right sibling node.

In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling node.



Deleting a leaf key (31)If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

Deleting 30 results in the above case.

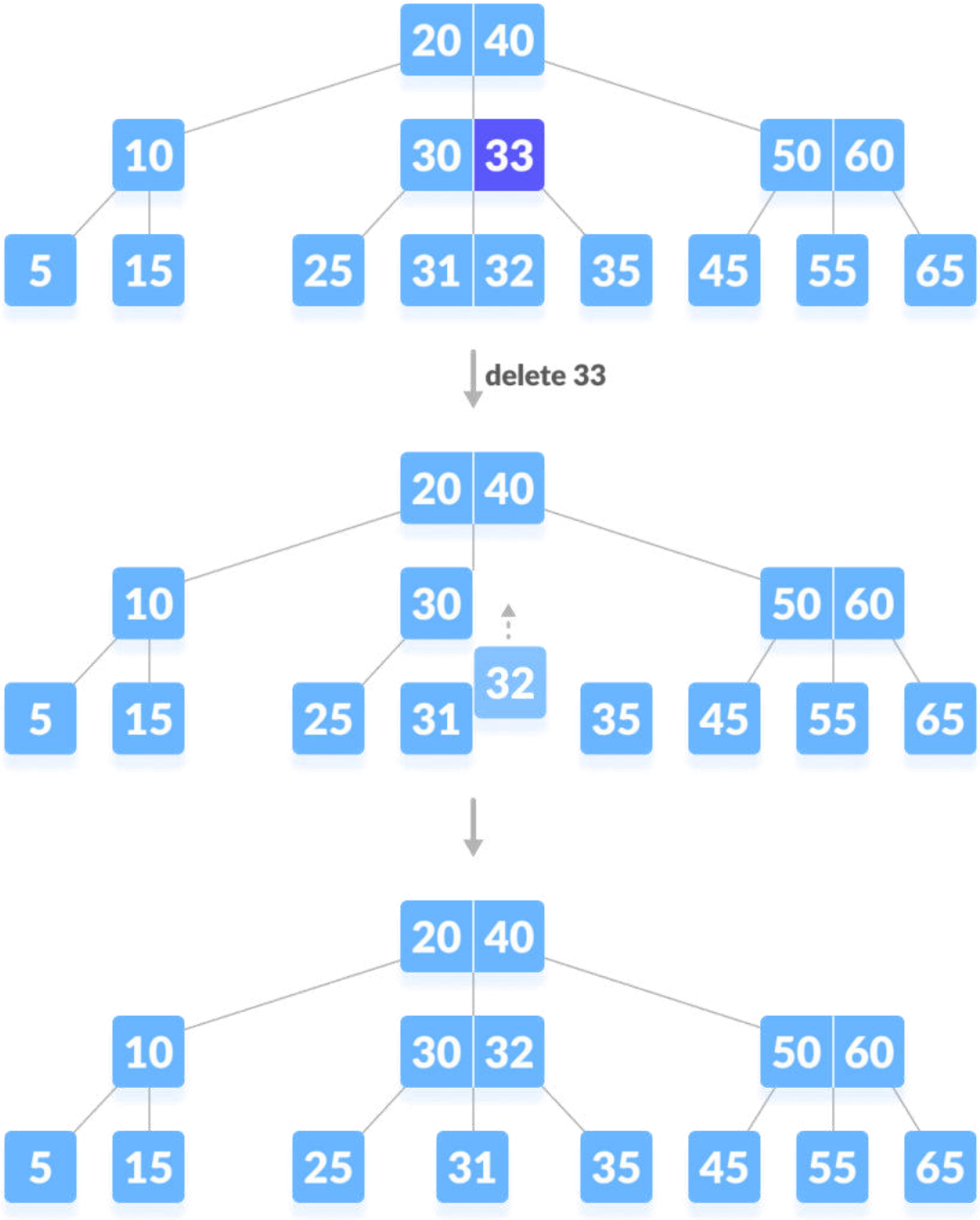


Delete a leaf key (30)

Case II

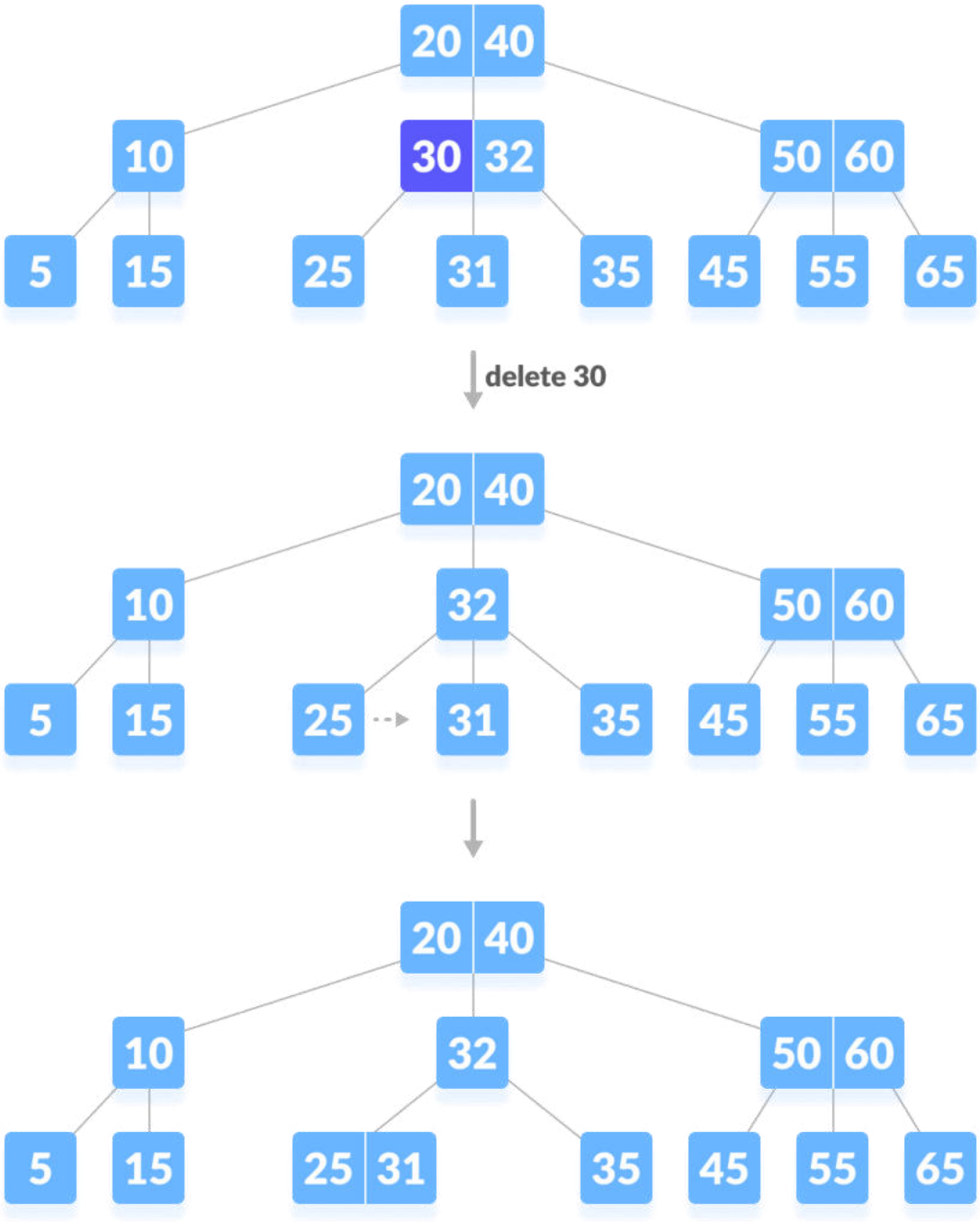
If the key to be deleted lies in the internal node, the following cases occur.

1. The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.



Deleting an internal node (33)

2. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
3. If either child has exactly a minimum number of keys then, merge the left and the right children.

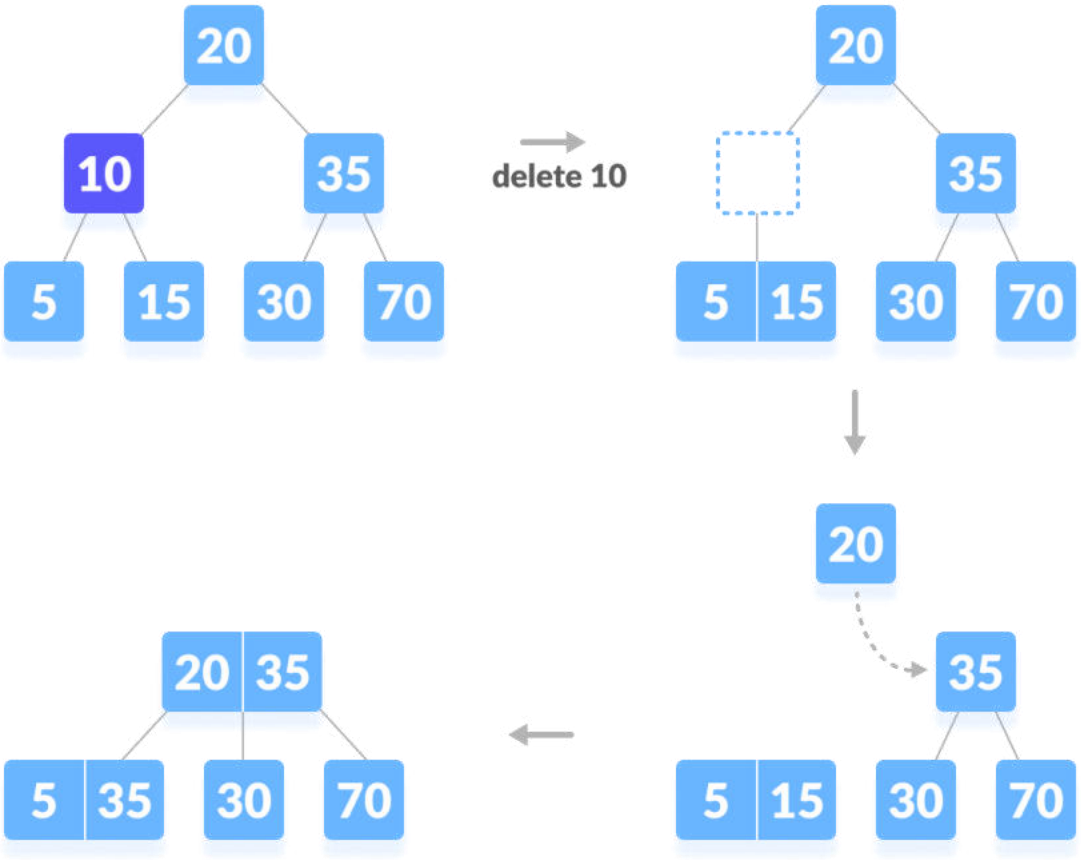


Deleting an internal node (30)After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

Case III

In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).



Deleting an internal node (10)

Advantages:

1. **Balanced Tree**: All leaves are at the same level, ensuring consistent performance.
2. **Efficient Searching**: Fast search operations with O(log n) time complexity.
3. **Disk-Friendly**: Minimizes disk I/O by storing multiple keys per node.
4. **Efficient Insertions/Deletions**: Maintains balance and performance with O(log n) complexity.
5. **Handles Large Data**: Good for large datasets and storage systems.

Disadvantages:

1. **Complex Implementation**: More complex to implement than simpler trees.
2. **Memory Overhead**: Uses more memory per node due to multiple keys and pointers.
3. **Not Ideal for In-Memory**: Less efficient for in-memory data compared to other trees.
4. **Node Operations**: Splitting and merging nodes can be complex and costly.
5. **Small Node Size Issues**: Small nodes can lead to decreased performance.

Applications

- **Databases**: For indexing and speeding up queries.
- **File Systems**: To organize and access files and directories.
- **Key-Value Stores**: To manage and retrieve data efficiently.
- **Operating Systems**: For managing virtual memory and page tables.
- **Search Engines**: For indexing and fast retrieval of search results.
- **Network Routers**: To manage routing tables for packet delivery.
- **Data Warehousing**: To efficiently organize and query large datasets.
- **Geospatial Systems**: For indexing and querying spatial data.

UNIT – II

Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications.

Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications.

Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull.

Heaps :-

A "heap" is a complete binary tree in which the nodes are organized based on their data entry values.

Two types of heap structures.

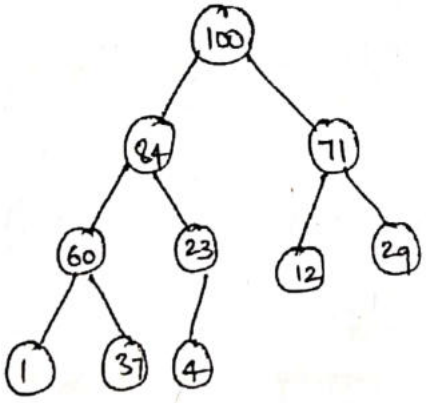
(i) Max - heap (ii) Min - heap.

(i) Max - heap :- A max-heap has property known as "the "heap order property," that each non-leaf node 'V', the 'V' value is greater than its two children.

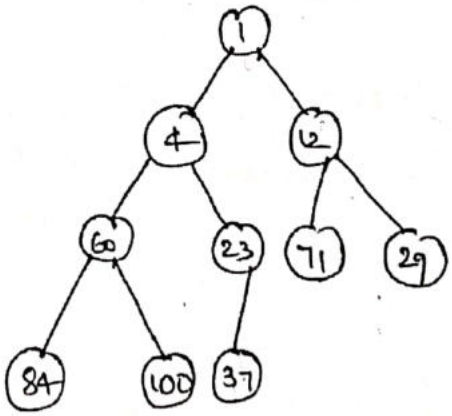
(ii) Min - heap :- In Min-heap, for each non-leaf node 'V', the value in 'V' is smaller than the value of its two children.

Example :-

(i) Max - heap



(ii) Min - heap



NOTE :- (1) The heap (Min-heap (or) Max-heap) allows insertion, deletion of nodes.

(2) when a new value is inserted (or) removed from existing value the "heap order property" (or) heap Shape property (A complete binary tree) must ensured

Example:- Consider the following set of values and use them by adding one value at a time, in the order listed.

30, 63, 2, 89, 16, 24, 19, 52, 27, 9, 4, 45.

(a) Max - heap (b) Min - heap

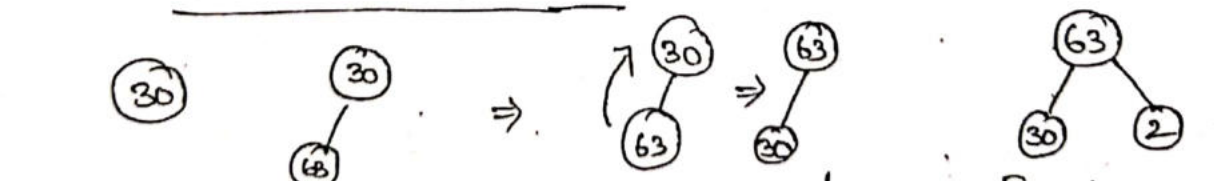
Sol:- Max - heap:- (Non leaf node is greater than its children)

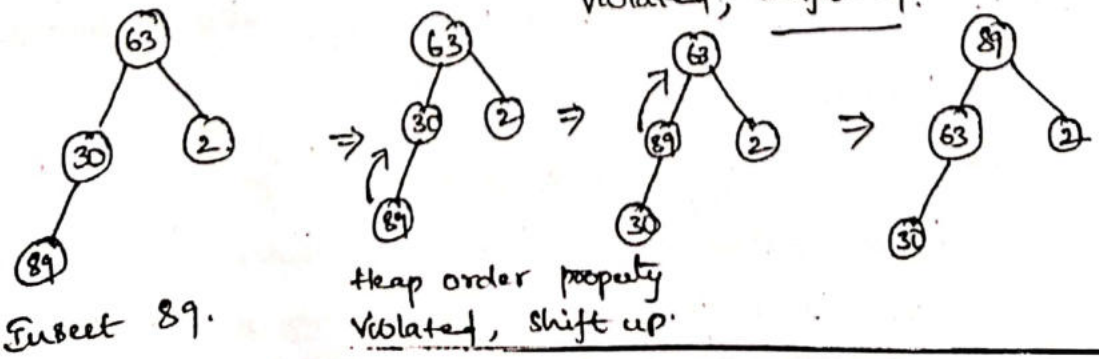
NOTE:- To restore the heap order property, the new value has to move up along the path, and going up until no violation of heap-order property occurs.

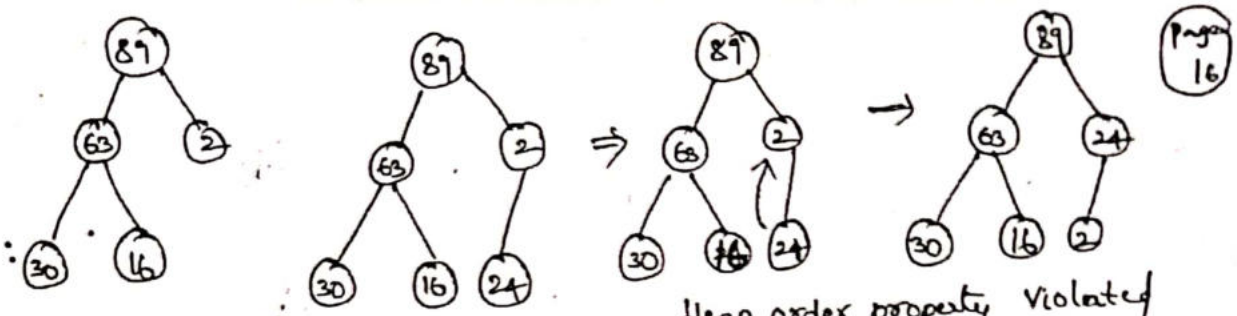
The upward movement of the newly inserted entry by means of swaps is conventionally called "Up-heap bubbling."

Elements = 30, 63, 2, 89, 16, 24, 19, 52, 27, 9, 4, 45

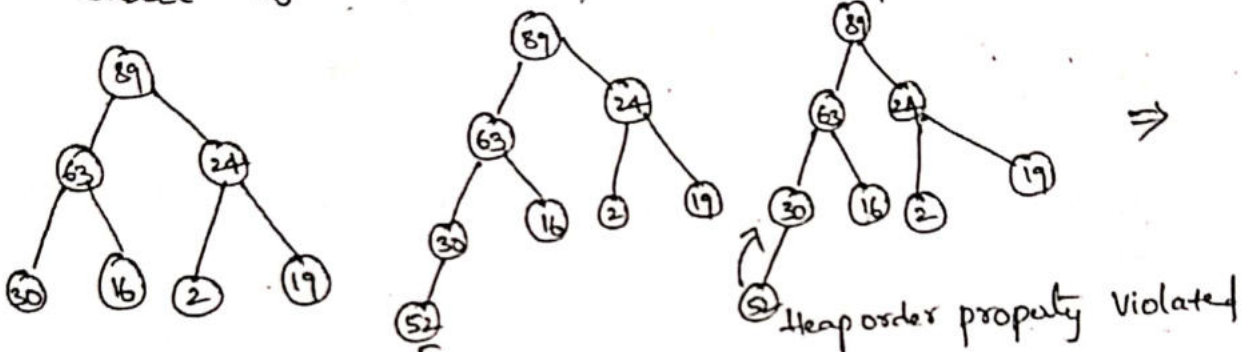
Insert in to heap structure by element by element, one element at a time.



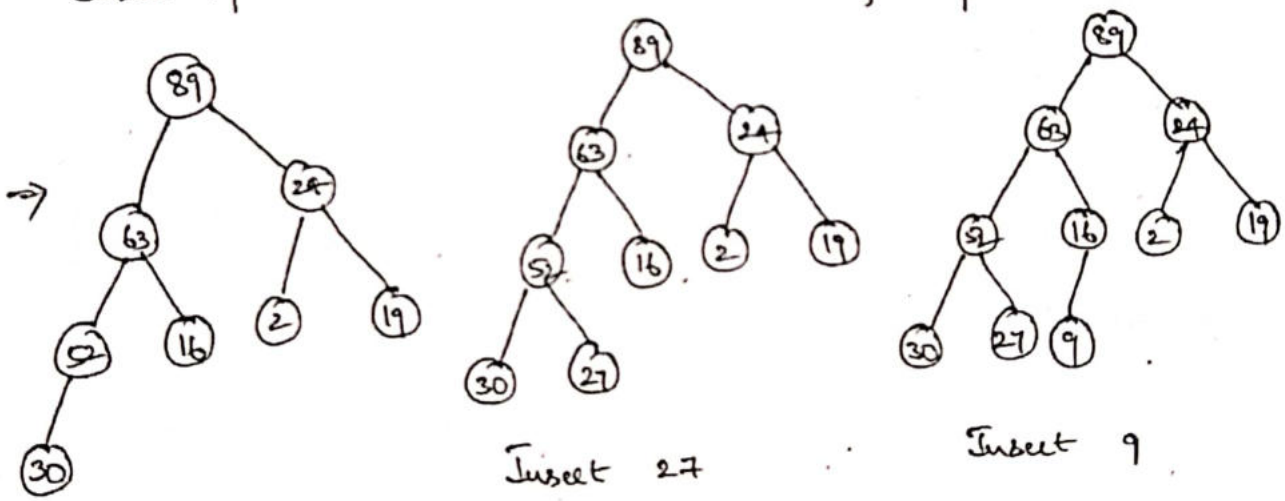




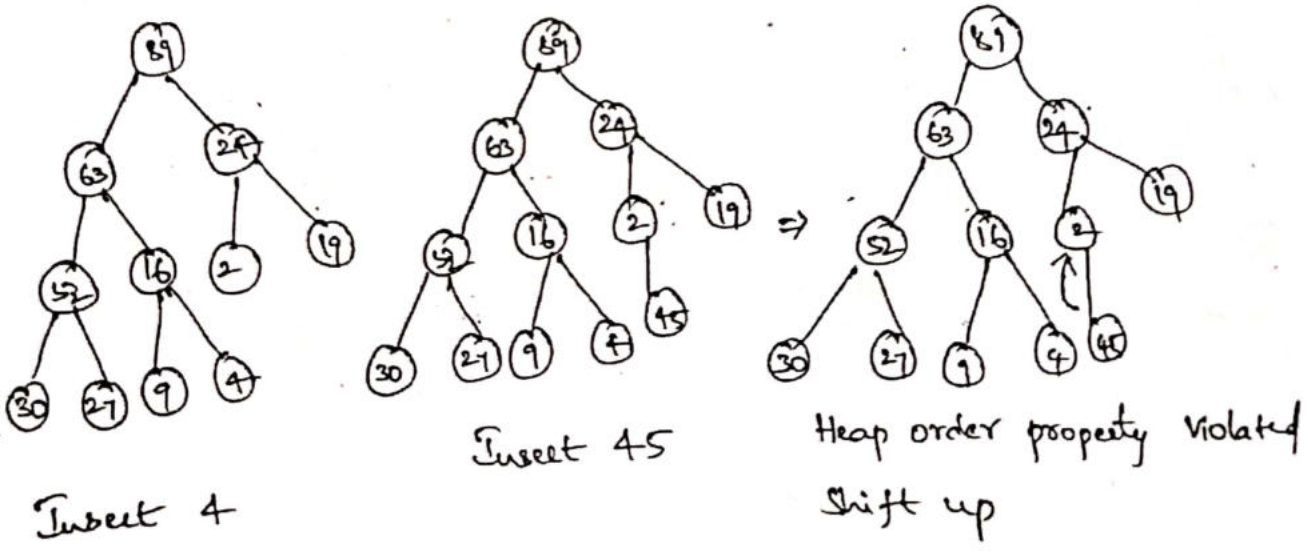
Insert 16 Insert 24 Shift up



Insert 19 Insert 52 Shift up.

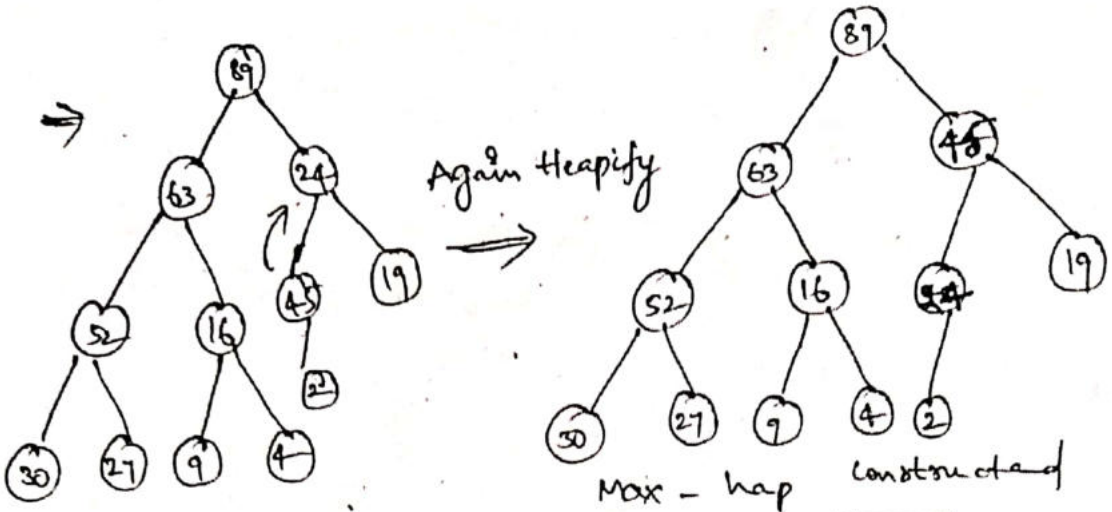


Insert 27 Insert 9



Insert 4 Insert 45 Heap order property Violated

Shift up

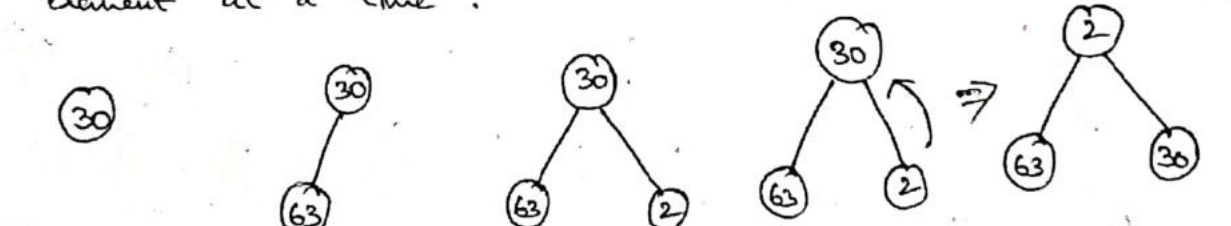


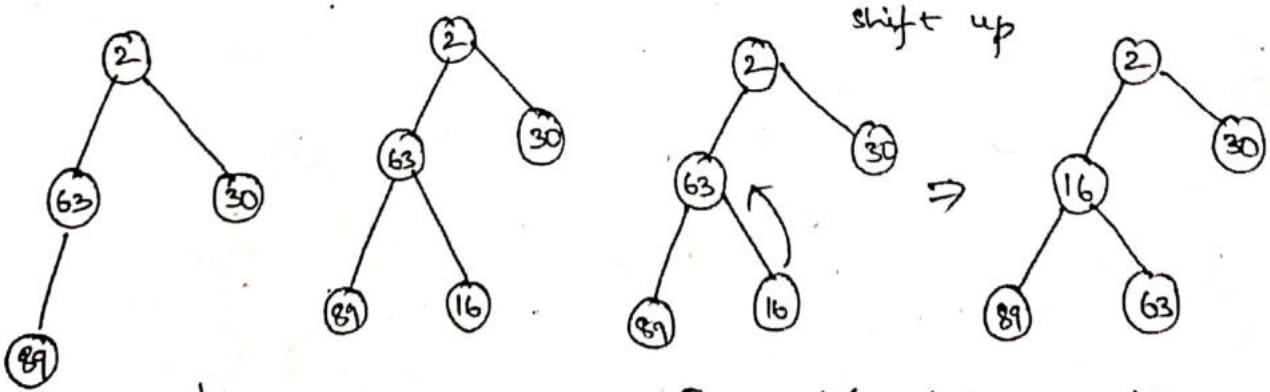
Max - heap constructed

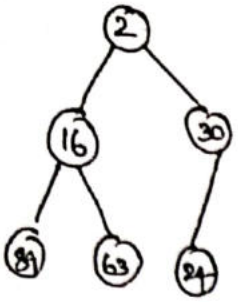
Min - heap: (Non leaf node is less than to its children)

30, 63, 2, 89, 16, 24, 19, 52, 27, 9, 4, 45.

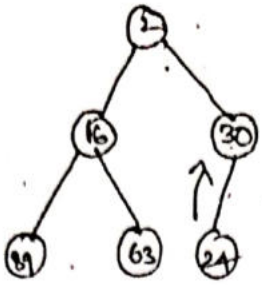
Insert in to Min heap structure element by element, one element at a time.



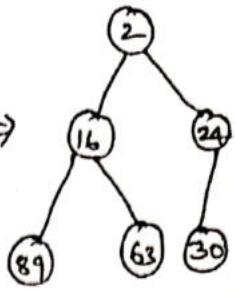


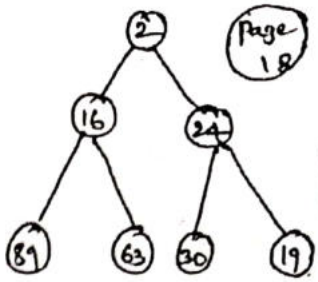


Insert 24

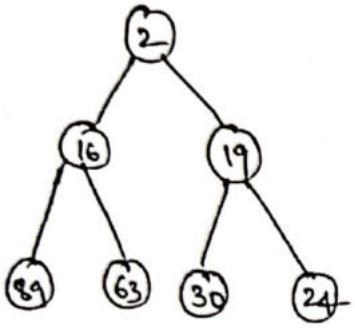


Min heap property Violated Shift up

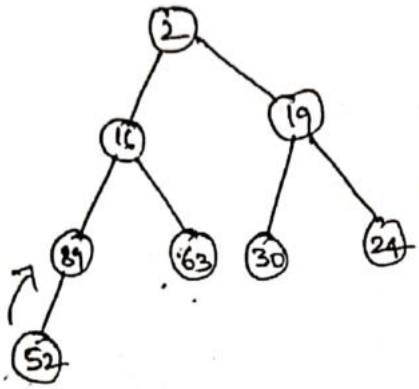




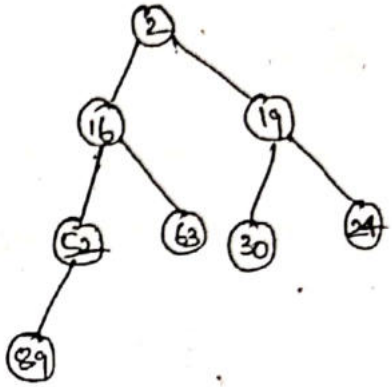
Insert 19

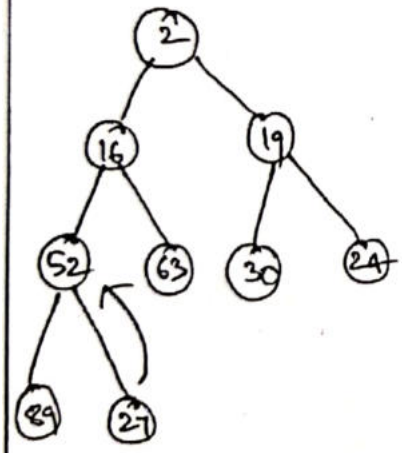


Minheap property Violated Shift up

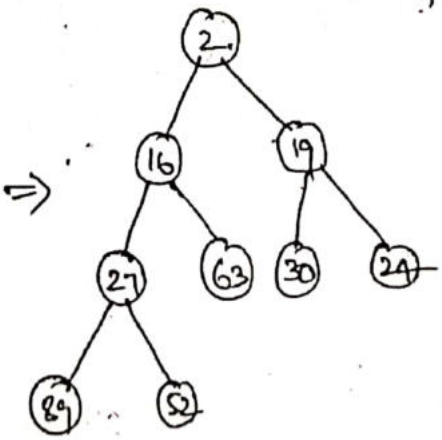


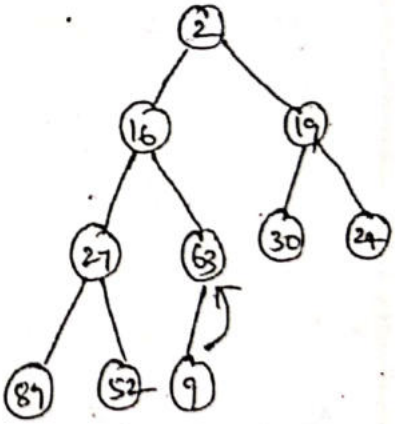
Insert 52, Min heap property Violated Shift up



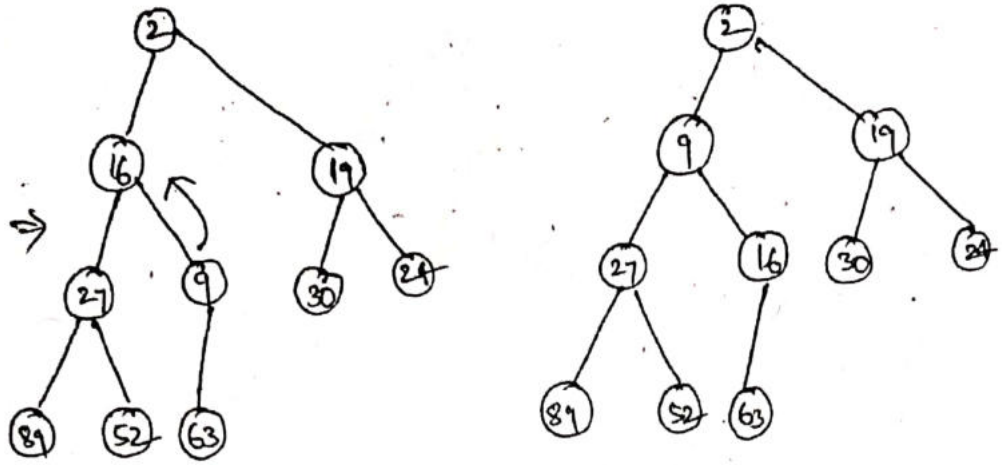


Insert 27, Min heap property Violated, Shift up

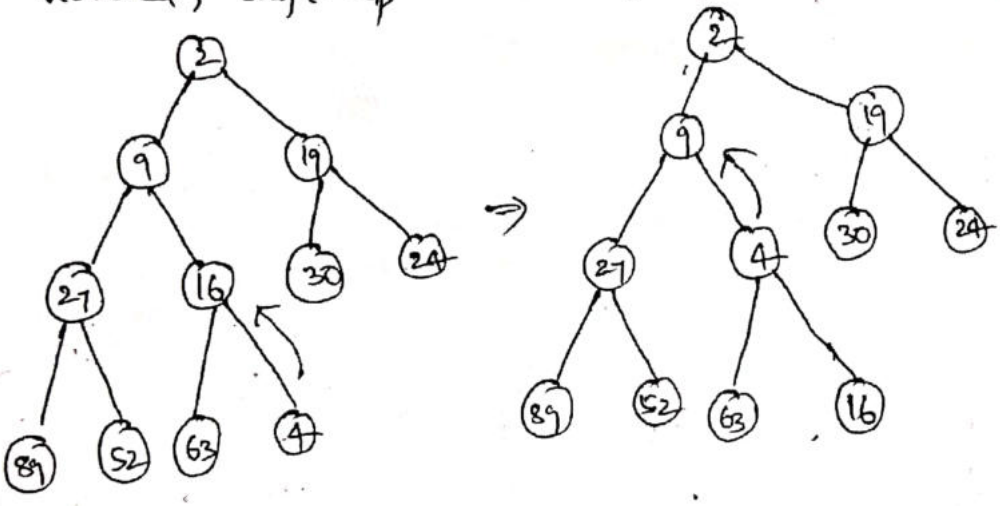




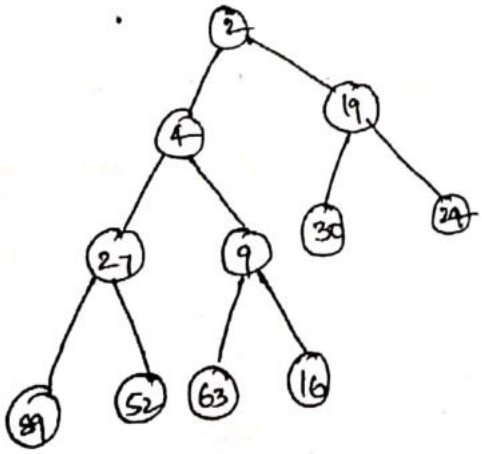
Insert 9, Min heap property Violated, Shift up



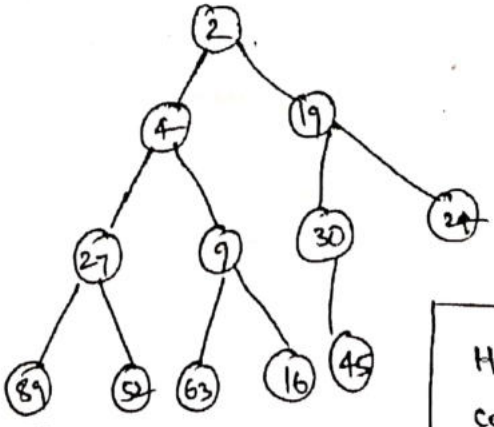
Again Min heap property violated, shift up



Insert 4, Min heap property violated, shift up



Again Min heap property violated, shift up



Insert 45,

Heap construction completed, all elements inserted.

Heap Operations :-

The heap is a specialized structure with limited operations.

(i) Insert a new value in to a heap.

(ii) Extract and remove the node value from the heap.

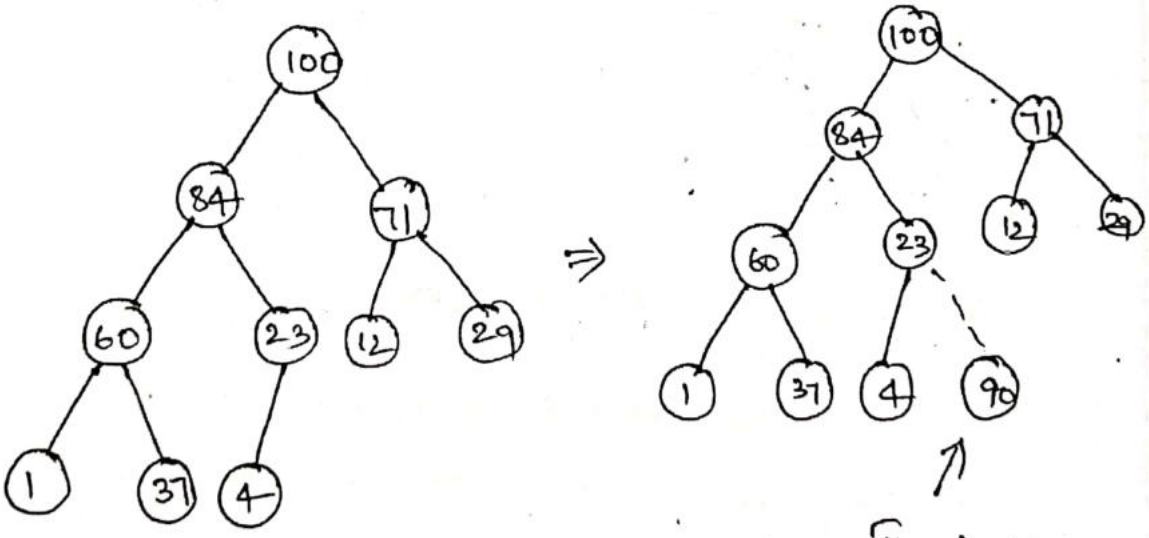
The above operations applicable both for Min-heap and Max-heap.

For Maximum Heap construction and its operations program refer page : 73

Insertion :

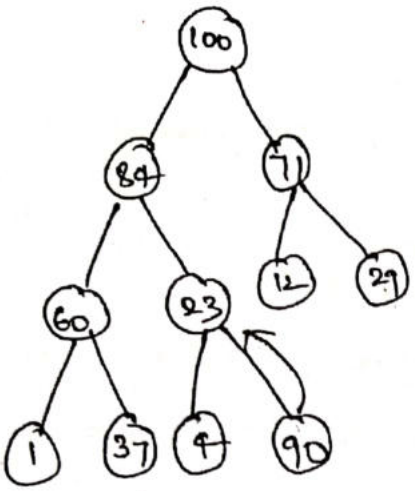
When a new value is inserted in to a heap, the heap order property and the heap shape property (A complete binary tree) must be maintained.

For Example, consider the following Max-heap.

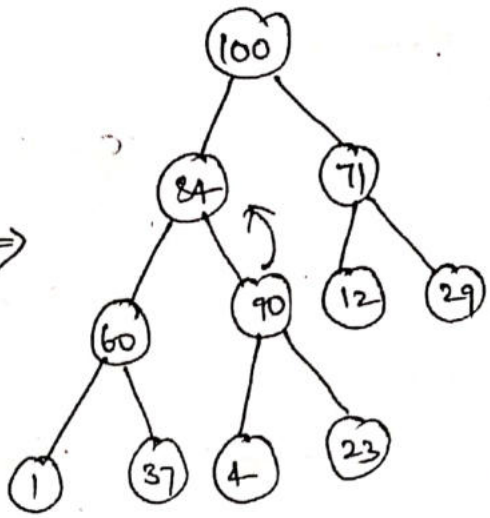


Insert a new node for 90

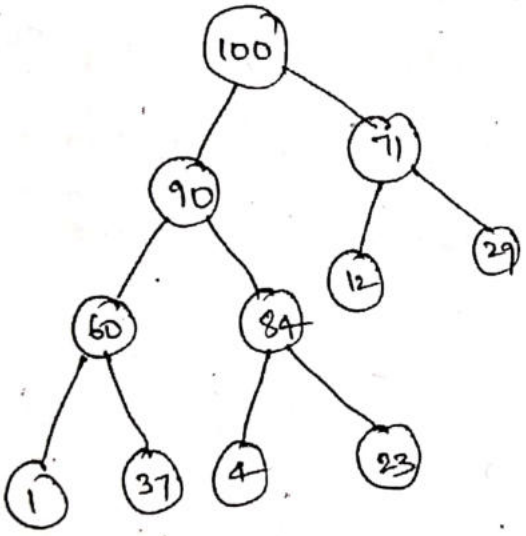
Insert as a child of "23" by ensuring Heap shape property



Max-heap property violated, shift-up



Again Max-heap property violated, shift-up



Extraction and Deletion from Heaps:-

The standard extraction and deletion operation on heap is to delete the element present at the root node of the heap.

If it is a Max heap, the standard deletion operation will delete the maximum element.

If it is a Min-heap, it will delete minimum element.

Deletion process :-

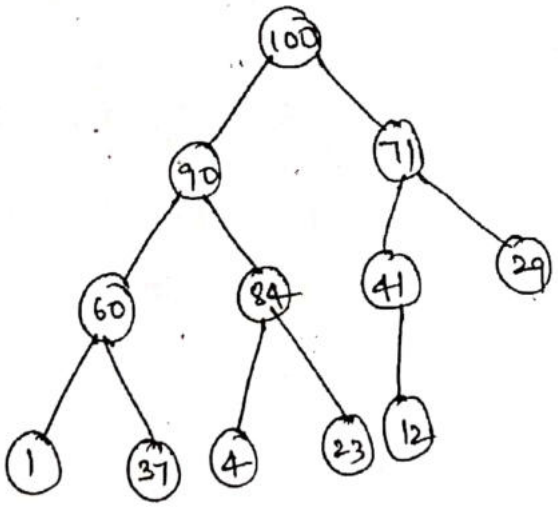
Step 1) Replace the root (or) element to be deleted by the last element.

Step 2) Delete the last element from the heap.

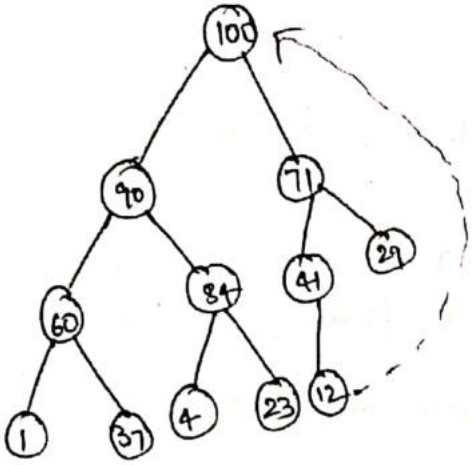
Step 3) The last element is now placed at the position of the root node.

If heap property not followed, and violated perform heapify (Shift up) operation to adjust the heap.

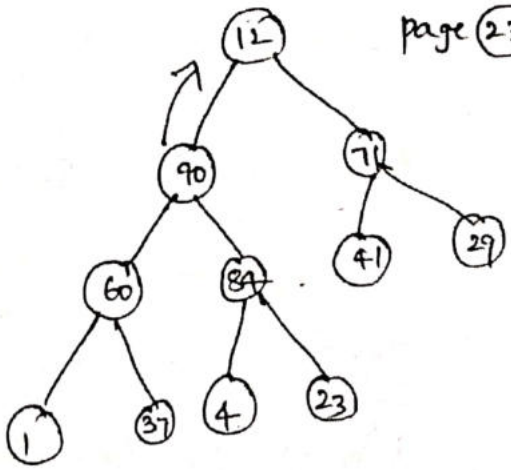
For Example from the following heap, Delete root element 100



Delete 100 from above Max-heap



⇒

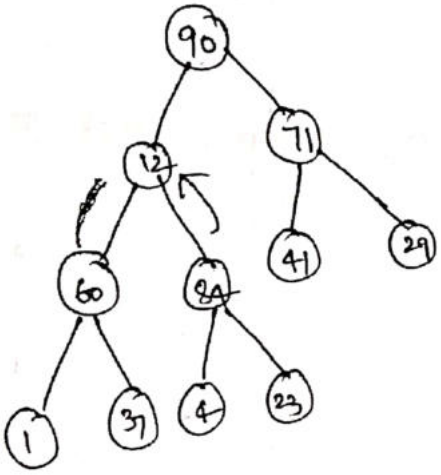


Copy the last item to the root

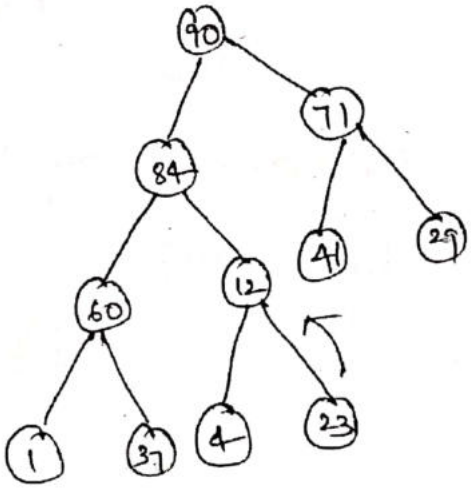
100 deleted.

Adjust the heap,

Heap property violated.



⇒

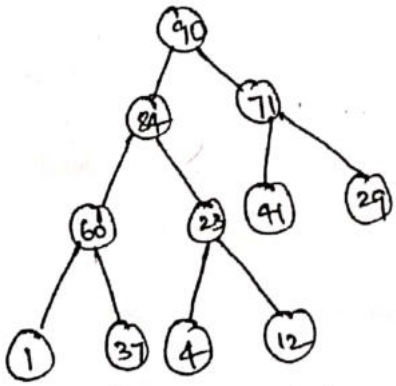


Again Adjust the heap.

Heap property violated

Again Adjust the heap.

Heap property violated.



NOW Heap principle adjusted. Deletion Successful.

Heap Sort :-

→ The simplicity and efficiency of the heap structure can be applied for sorting problem.

→ The heapsort algorithm builds a heap from a sequence of unsorted values and then extracts the items from the heap to create a sorted sequence.

→ The heapsort algorithm is very efficient and only requires $O(n \log n)$ time in worst case.

Heap Sort steps :-

Step 1) : Construct max-heap for a given list of elements.

Step 2) : The tree satisfies Max-heap property, then the largest item is started at the root node.

Step 3) : Remove the root element and put at the end of the array, put the last item of the tree (heap) at the vacant place.

Step 4) :- Reduce the size of the heap by 1.

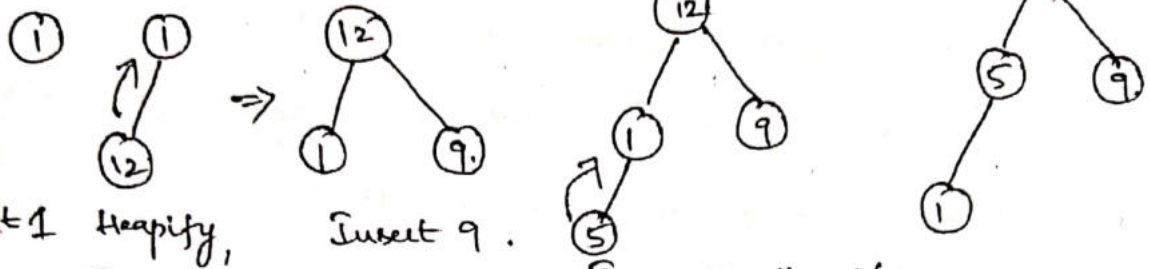
Step 5) :- Heapify the root element again, so that we have the highest element at root.

Step 6) :- This process is repeated until all the items of the list are sorted.

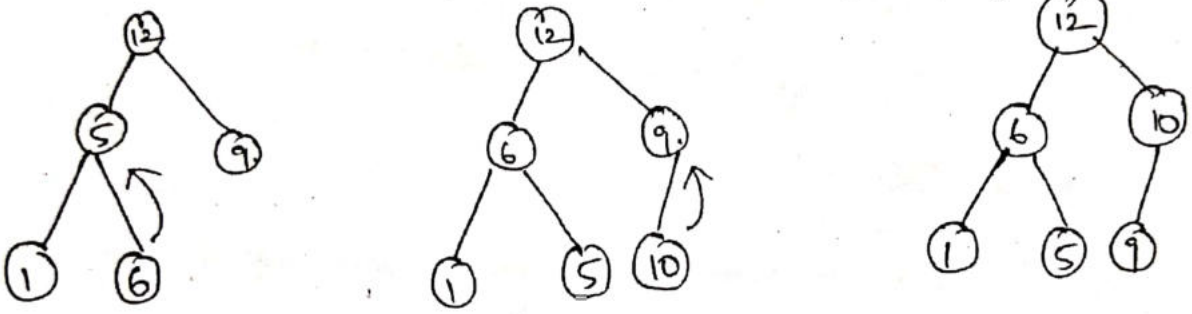
Example:- Sort the element 1, 12, 9, 5, 6, 10 using Heap sort.

Step 1) Construct Max-Heap for given elements. (Page 25)

1, 12, 9, 5, 6, 10.

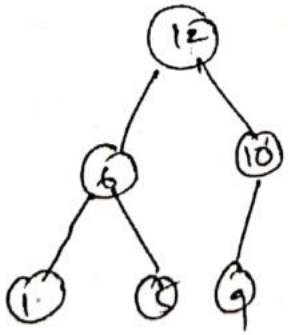


Insert 1 Heapify, Insert 12 Insert 9. Insert 5, Heapify.

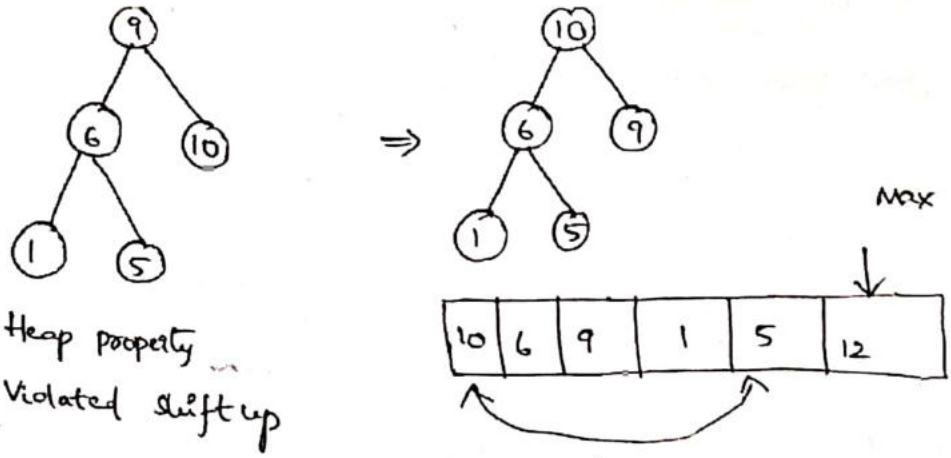


Insert 6, Heapify. Insert 10, Heapify

Step 2):- The constructed Satisfies Max-Heap property

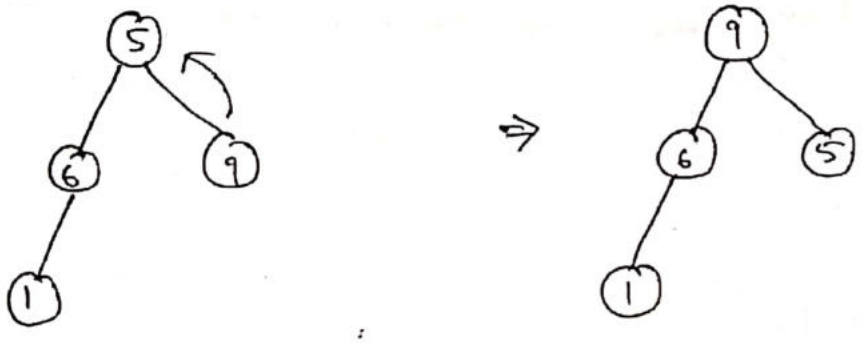


Step 3): Remove root node of the tree, and put the end of the array, put the last item at the vacant place.

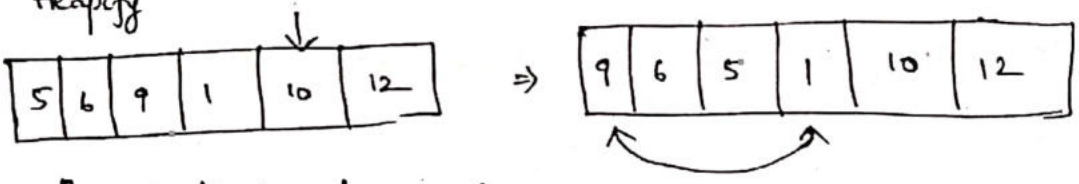


Heap property Violated shift up

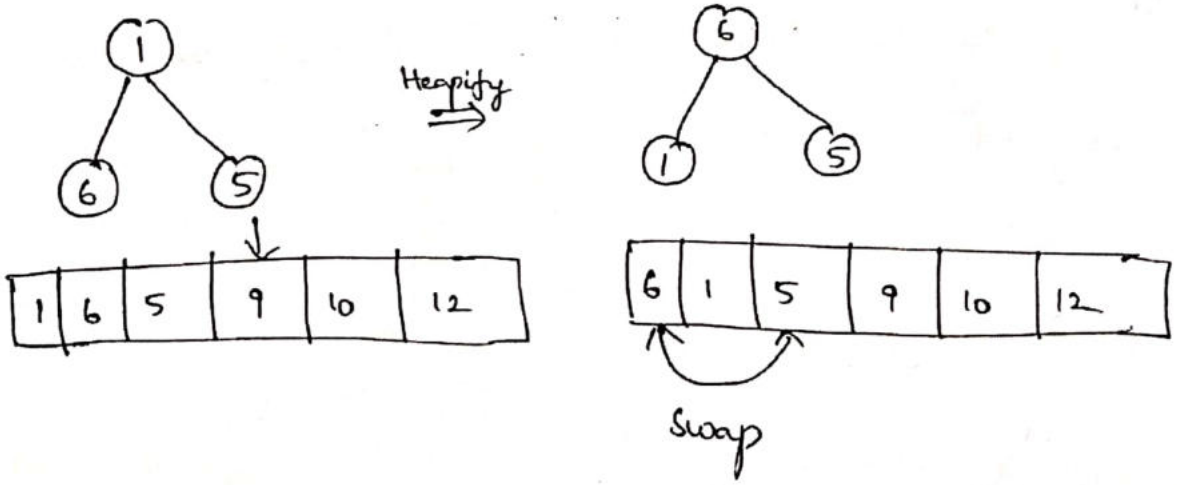
Remove Root from heap, and place at next last element



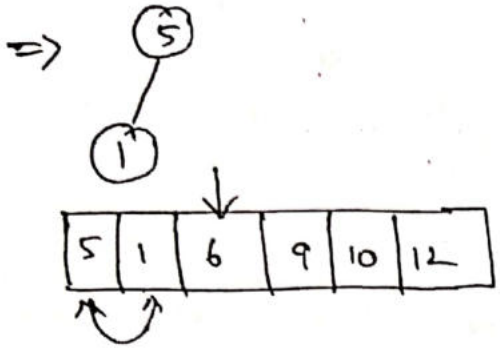
Heapify



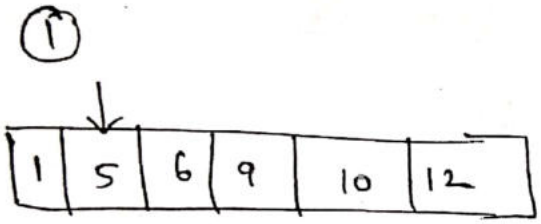
Remove Root from heap, and place at next last element



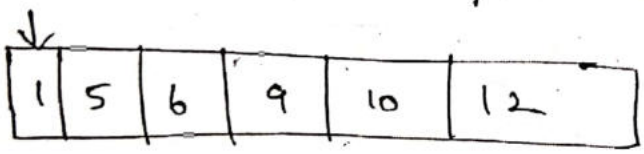
Remove Root from heap, and place at next last element



Remove Root from heap, and place at next last element.



Remove Root from heap, and place at next last element.



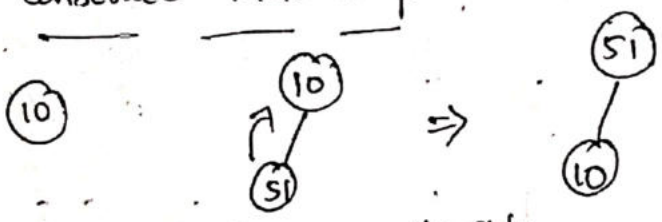
Finally in the above array all elements are sorted

Sorting order = 1, 5, 6, 9, 10, 12

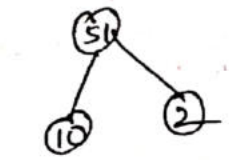
Ex:- Sort the following list of elements using Heap sort.

10, 51, 2, 18, 4, 31, 13, 5, 23, 64, 29.

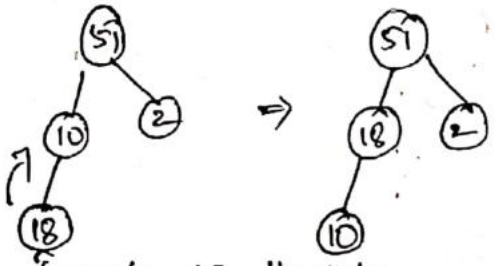
Construct Max-heap:-



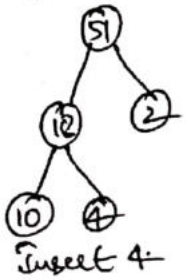
Insert 10 — Insert 51, Heapify



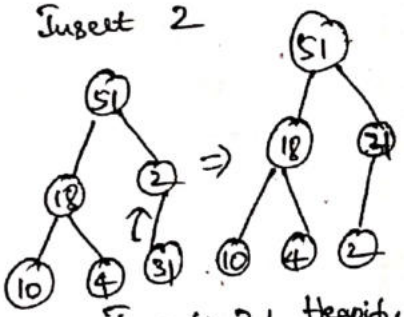
Insert 2



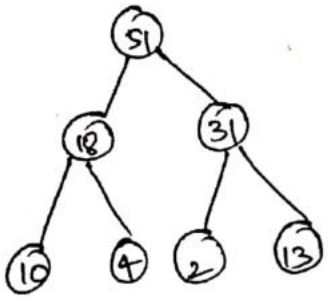
Insert 18, Heapify.



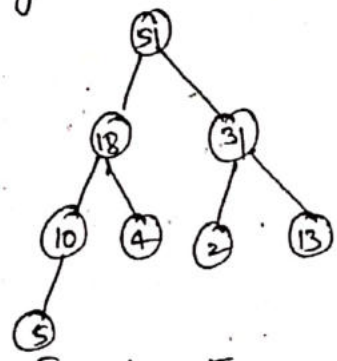
Insert 4



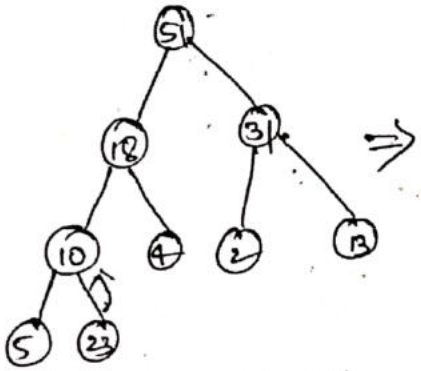
Insert 31, Heapify



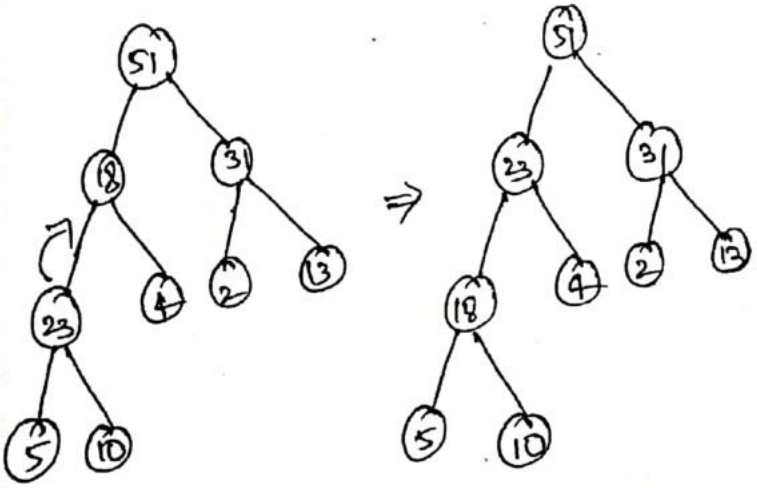
Insert 13



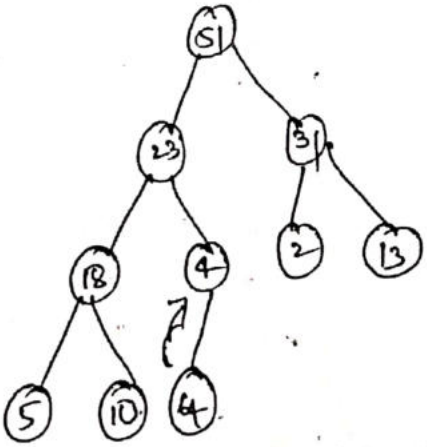
Insert 5



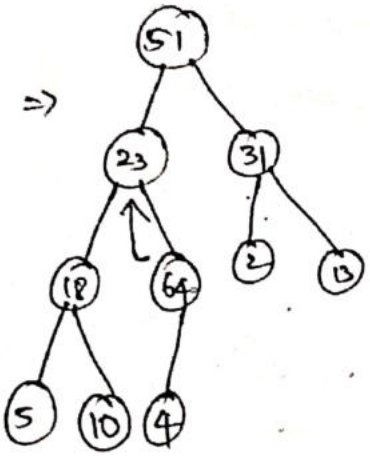
Insert 23, Heapify.

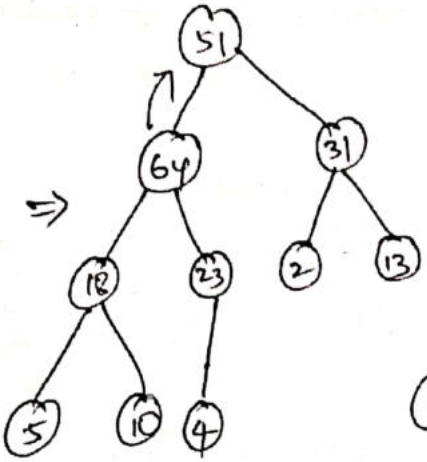


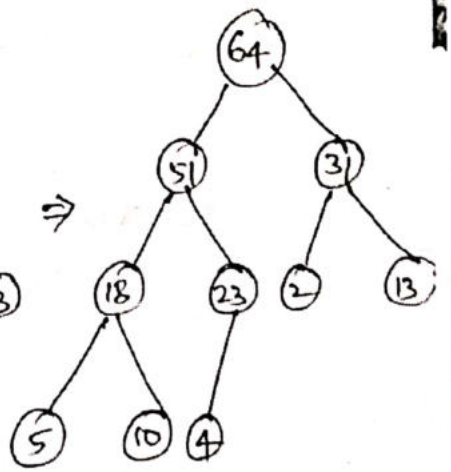
Again Heapify.



Insert 64, Heapify.

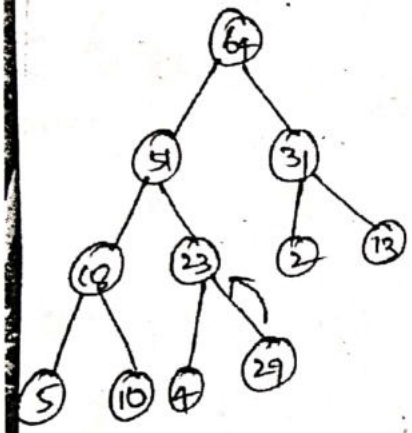


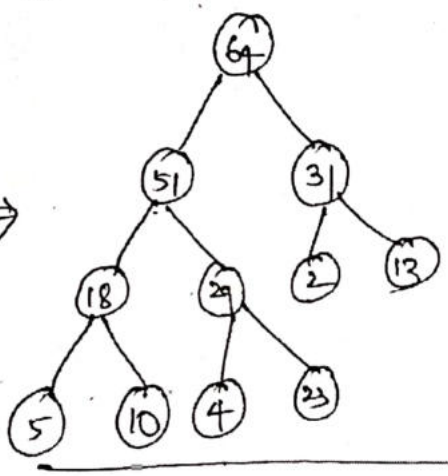




Again Heapify.

Again Heapify.

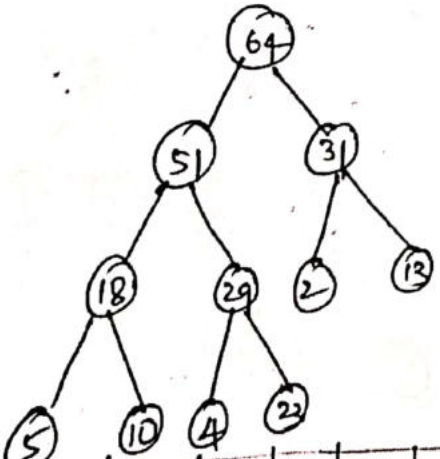




Insert 29, Heapify

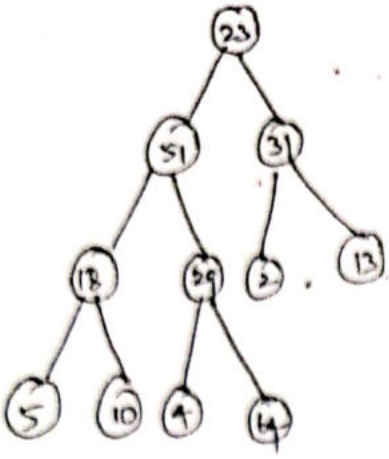
Max heap constructed.

Construct Sorting list with Max heap.



Original Max Heap

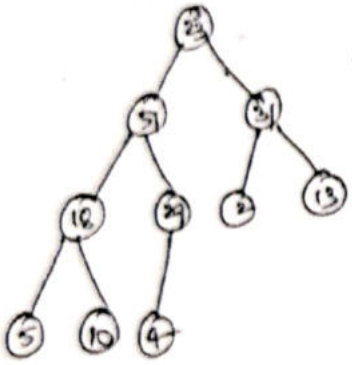
64	51	31	18	29	2	13	5	10	4	23



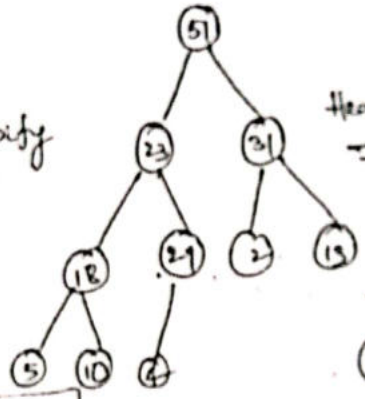
64 23

23	51	31	18	29	2	13	5	10	4	64

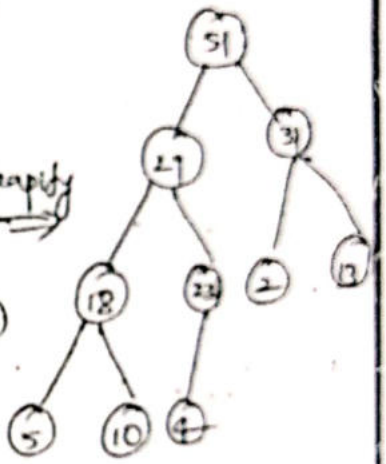
Swap first and last elements in the heap.



Heapify ⇒



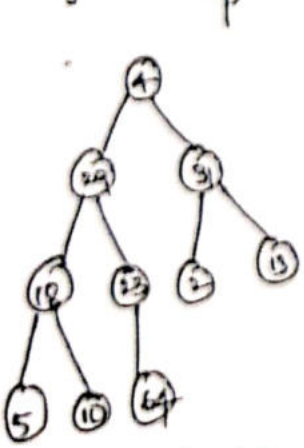
Heapify ⇒



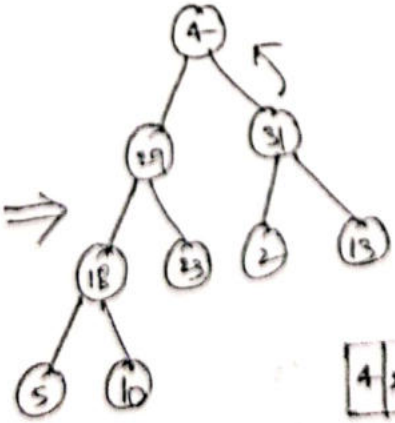
4	51	29	31	18	23	2	13	5	10	64

Remove the last element from heap.

51	29	31	18	23	2	13	5	10	4



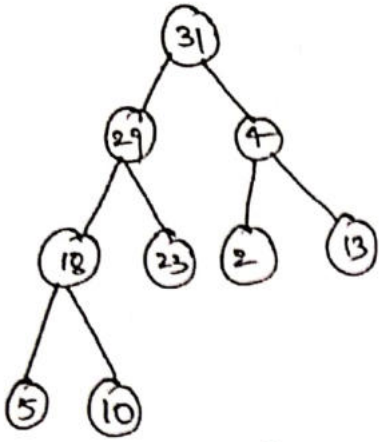
⇒



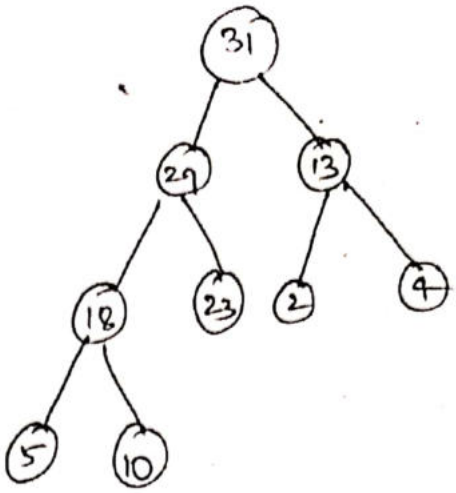
Heapify ⇒

4	29	31	18	23	2	13	5	10	51	64

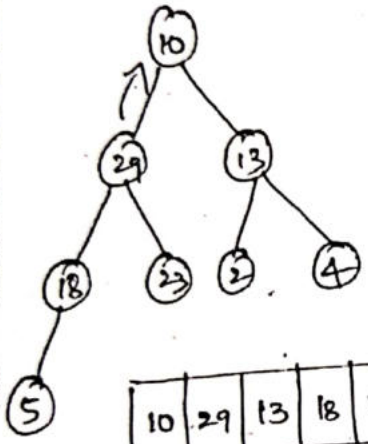
Swap 51 with 4 and remove 51 from heap.



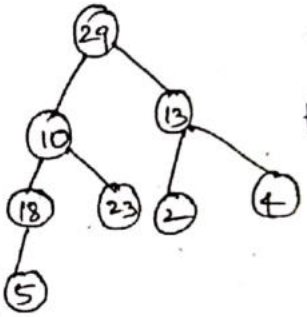
Heapify



31	29	13	18	23	2	4	5	10	51	64



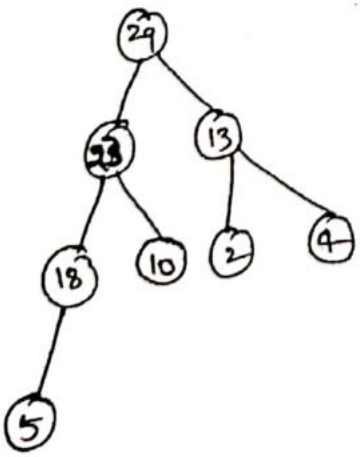
Heapify



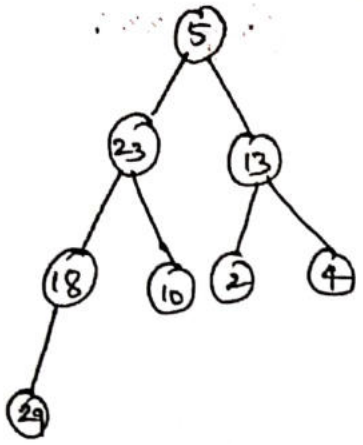
Again Heapify

10	29	13	18	23	2	4	5	31	51	64

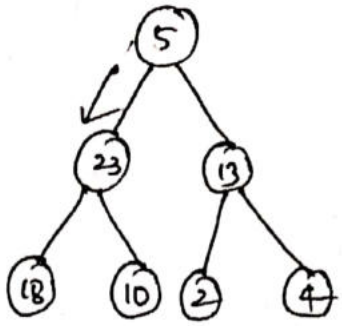
Swap 10 with 31, Remove 31 from heap.



29	23	13	18	10	2	4	5	31	51	64



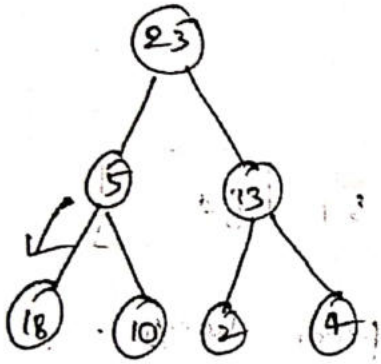
⇒



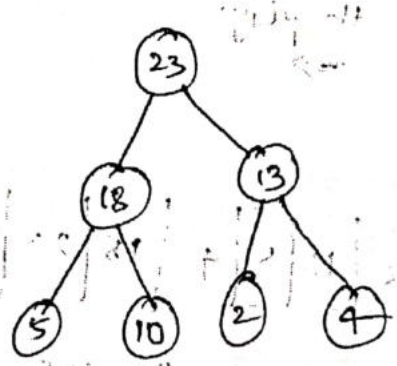
Heapify ⇒

5	23	13	18	10	2	4	29	31	51	64

Swap 29 with 5, Remove 29 from heap.

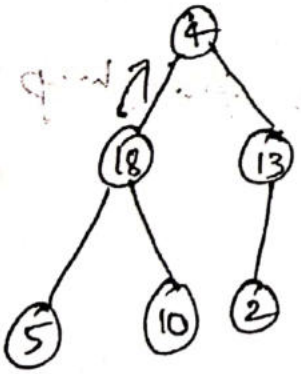


⇒

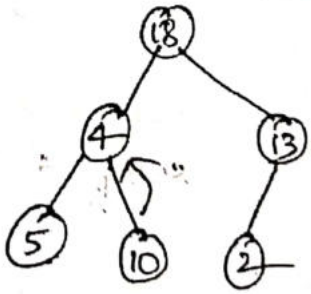


Again Heapify

23	18	13	5	10	2	4	31	51	64



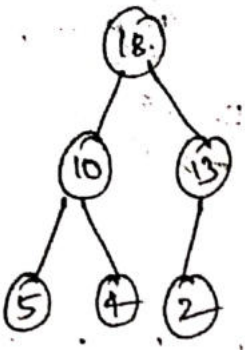
Heapify ⇒



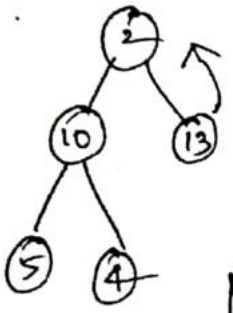
Heapify ⇒

4	18	13	5	10	2	23	31	51	64

Swap 23 with 4, Remove 23 from heap.



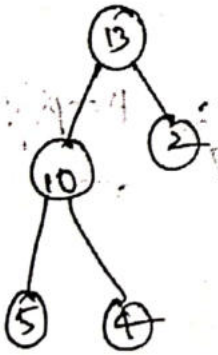
18	10	13	5	4	2	23	31	51	64



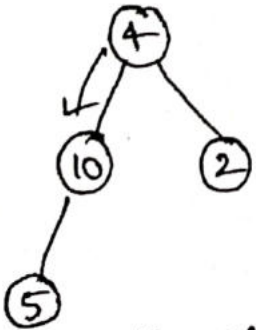
Heapify ⇒

2	10	13	5	4	18	23	31	51	64

Swap 18 with 2 and Remove 18 from heap.

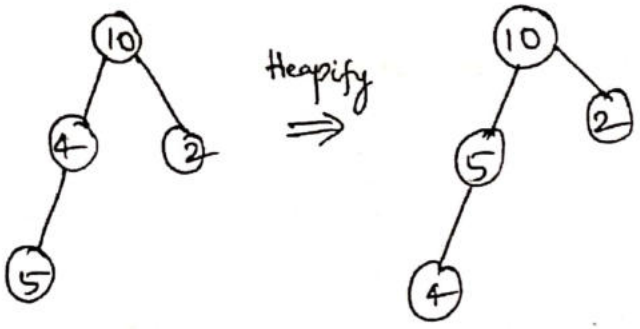


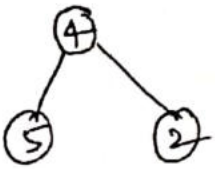
Swap 4 with 13, and remove 13 from heap



4	10	2	5	13	18	23	31	51	64

Heapify





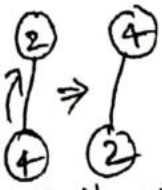
4	5	2	10	13	18	23	31	51	64

swap 10 with 4, remove 10 from heap.

Heapify.

2	4	5	10	13	18	23	31	51	64

swap 5 with 2, remove 5 from heap.



4	2	5	10	13	18	23	31	51	64

⇒ Heapify.

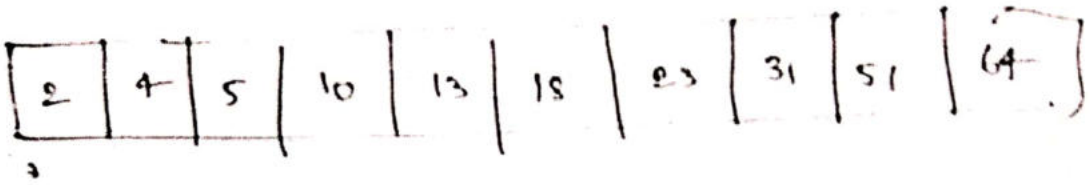
⇒ 

2	4	5	10	13	18	23	31	51	64

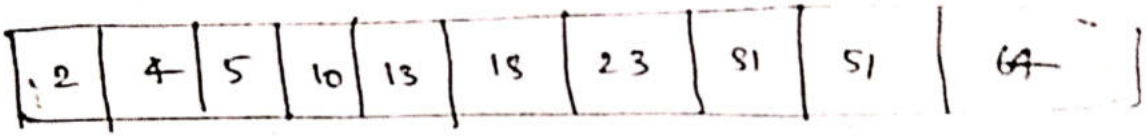
swap 4 with 2, remove 4 from heap.

Remove (2) from heap.

(2) ⇒ Heap Empty



Finally we get Sorted List



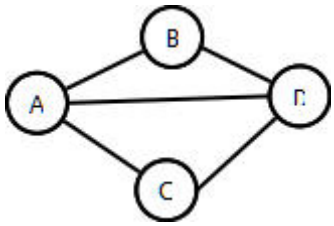
Graphs:

Definitions

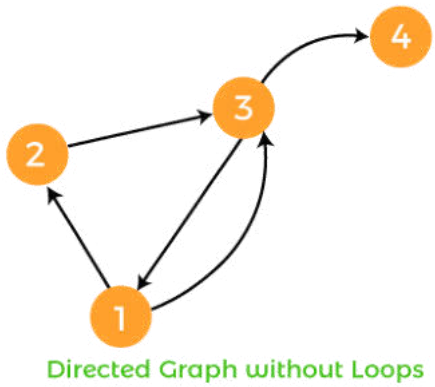
1. **Graph**: A graph G is defined as a pair G=(V,E) where:
    - V is a set of vertices (or nodes).
    - E is a set of edges, which are pairs of vertices representing connections between them.

Example:

- V = {A,B,C,D}
- E = {(A,B),(A,C),(A,D),(B,D),(C,D)}

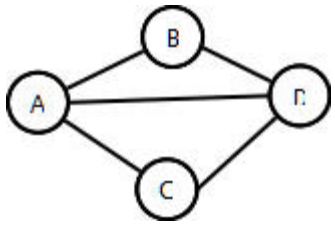


2. **Vertex (Node)**: A fundamental unit of a graph, represented as a point. Vertices are usually denoted by letters or numbers (e.g., V1,V2).
3. **Edge**: A connection between two vertices in a graph. Edges can be directed (having a direction) or undirected.
4. **Directed Graph (Digraph)**: A graph in which the edges have a direction, indicating a one-way relationship between vertices.

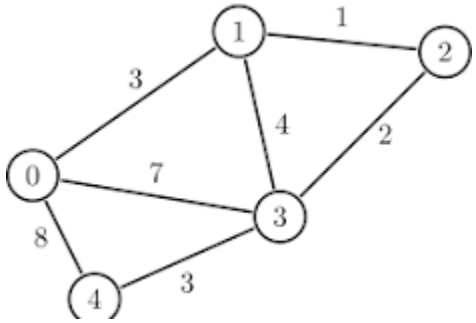


Directed Graph without Loops

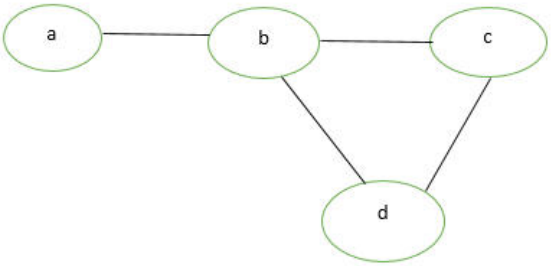
5. **Undirected Graph**: A graph in which edges do not have a direction, indicating a mutual relationship between vertices.

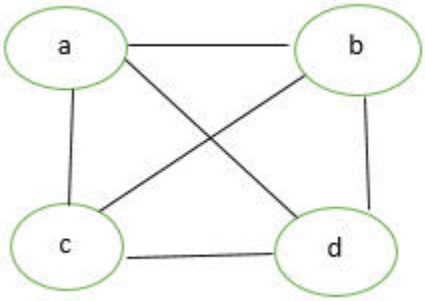


6. **Weighted Graph**: A graph where each edge is assigned a weight (or cost), representing the strength, distance, or capacity of the connection.

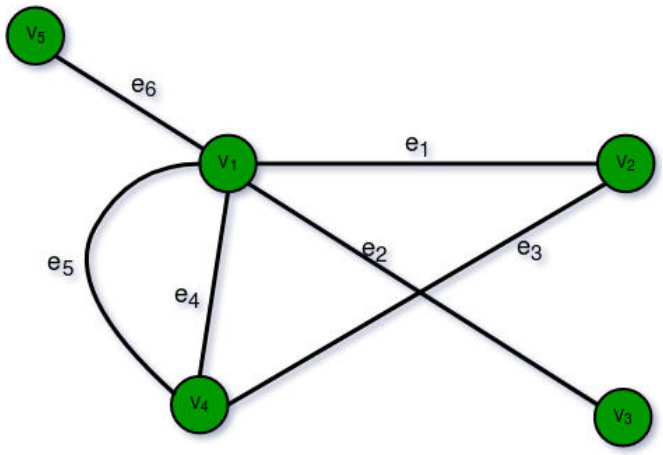


7. **Simple Graph**: A graph with no loops (edges connecting a vertex to itself) and no multiple edges between the same pair of vertices.

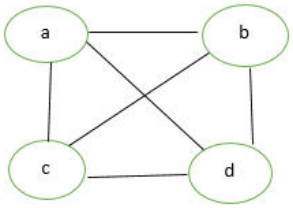


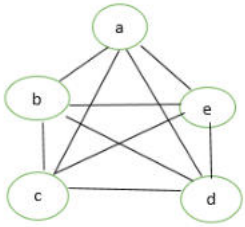


8. **Multigraph**: A graph that can have multiple edges (parallel edges) between the same pair of vertices.

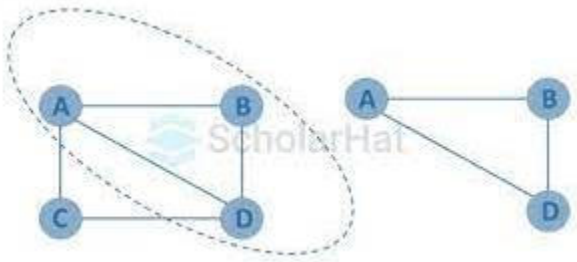


9. **Path**: A sequence of edges that connect a sequence of vertices without revisiting any vertex.
10. **Cycle**: A path that starts and ends at the same vertex, with no other repetitions of vertices and edges.
11. **Connected Graph**: A graph in which there is a path between every pair of vertices.
12. **Disconnected Graph**: A graph in which at least one pair of vertices does not have a path between them.
13. **Complete Graph**: A graph in which every pair of vertices is connected by a unique edge.

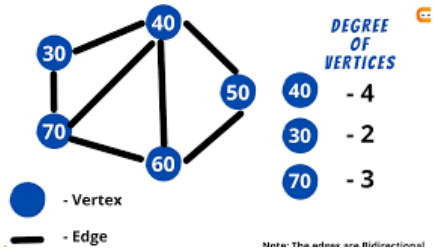


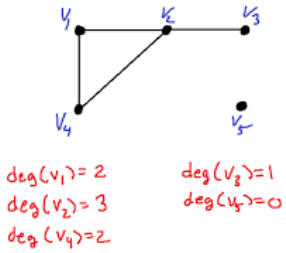


14. **Subgraph**: A graph formed from a subset of the vertices and edges of a larger graph.

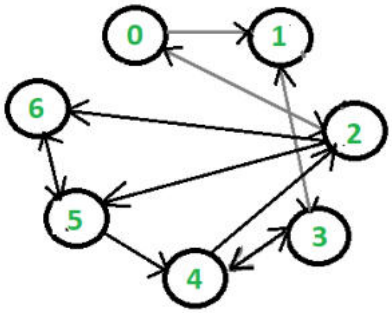


15. **Degree of a Vertex**: The number of edges connected to a vertex. In directed graphs, this includes:
    - **In-degree**: The number of incoming edges.
    - **Out-degree**: The number of outgoing edges.





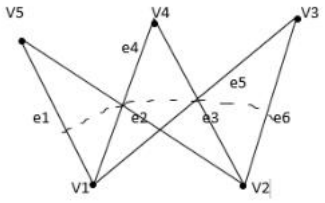
Directed Graph



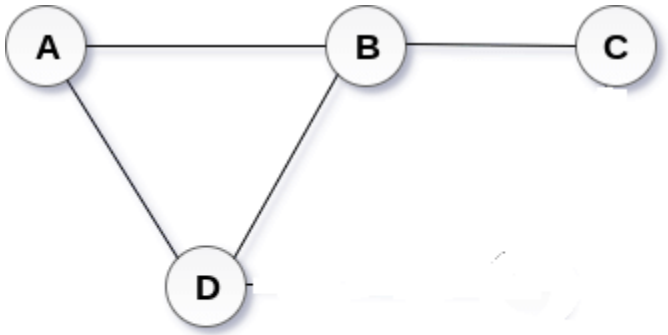

```
Output:
Vertex    In    Out
0         1    2
1          2    1
2          2    3
3          2    2
4          2    2
5          2    2
6          2    1
```

16. **Bipartite Graph**: A graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex in one set to a vertex in the other set.

V1(G) and V2(G) in such a way that each edge e of E(G) has one end in V1(G) and another end in V2(G). The partition V1 U V2 = V is called Bipartite of G. Here in the figure: V1(G)={V5, V4, V3} and V2(G)={V1, V2}



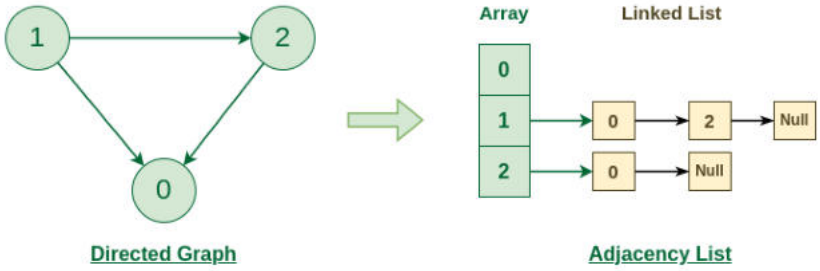
17. **Adjacency**: A relation where two vertices are connected by an edge.
18. **Adjacency Matrix**: A square matrix used to represent a graph, where the element at row i and column j indicates whether there is an edge between vertices i and j.



Undirected Graph

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

19. **Adjacency List**: A collection of lists or arrays used to represent a graph, where each list corresponds to a vertex and contains a list of adjacent vertices.



Graph Representation of Directed graph to Adjacency List

Graph Traversals:

Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It delves deeper into the graph along one path until it hits a dead-end (i.e., a vertex with no unvisited neighbors), then it backtracks and explores other paths.

Algorithm Steps:

1. **Start** at a given source vertex.
2. **Visit** the first unvisited neighboring vertex.
3. Repeat the process by going as deep as possible.
4. Backtrack when no unvisited vertices remain, and continue with other unvisited vertices.
5. Use a **stack** (either an explicit stack or recursive function call stack) to keep track of the vertices being explored.

Key Characteristics:

- Explores **deep into a graph** before visiting other vertices.
- Can be implemented recursively or iteratively using a stack.
- Used for applications like finding **connected components**, **topological sorting**, or **detecting cycles** in graphs.

Time Complexity:

- **O(V + E)**, where V is the number of vertices and E is the number of edges.

Example Use Cases:

- Solving mazes (exploring all possible paths).
- Detecting cycles in a graph.
- Topological sorting in a directed acyclic graph (DAG).

Breadth-First Search (BFS)

BFS explores the graph layer by layer, visiting all vertices at the current depth level before moving to the next level.

Algorithm Steps:

1. **Start** at a given source vertex.
2. **Visit** all its neighboring vertices (those directly connected by an edge).
3. For each visited neighbor, explore its unvisited neighbors in the same way.
4. Repeat the process level by level until all vertices are visited or no more neighbors remain.
5. Use a **queue** data structure to keep track of the vertices to be explored.

Key Characteristics:

- Explores vertices in order of distance from the start vertex (measured by the number of edges).
- Typically used to find the **shortest path** in unweighted graphs.
- Traversal guarantees visiting each vertex **once**.

Time Complexity:

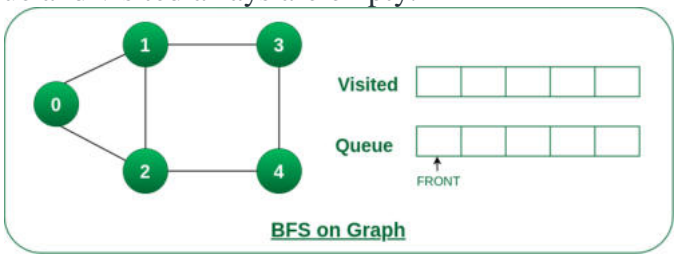
- **O(V + E)**, where V is the number of vertices and E is the number of edges.

Example Use Cases:

- Finding the shortest path in an unweighted graph.
- Level-order traversal in a tree.

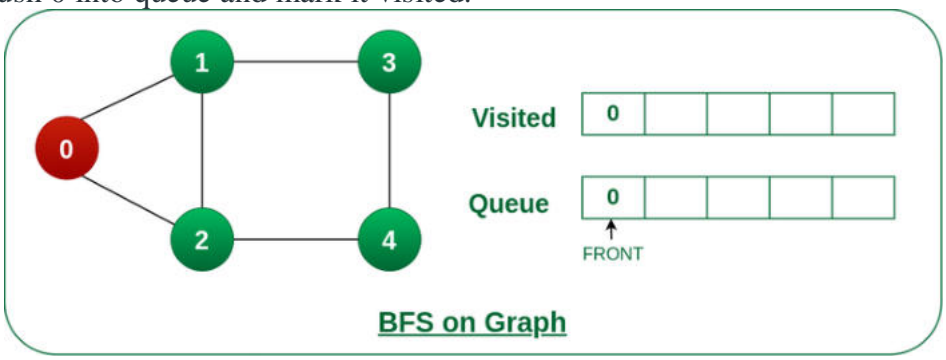
Example:

Let us understand the working of the algorithm with the help of the following example where the **source vertex is 0**.
Step1: Initially queue and visited arrays are empty.



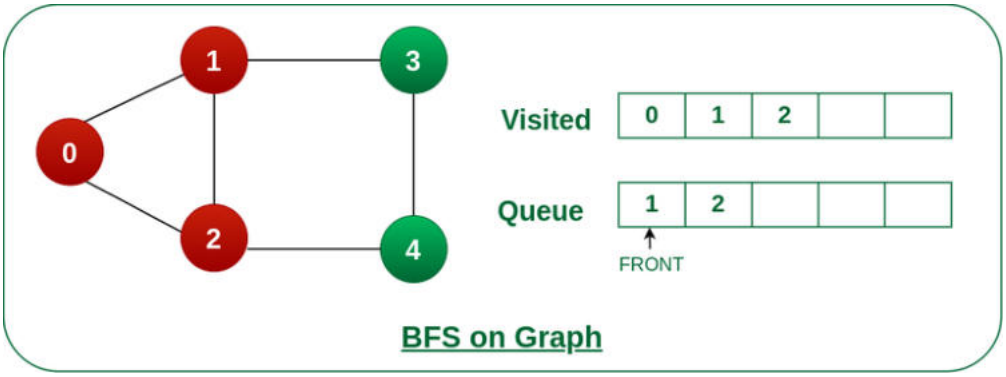
Queue and visited arrays are empty initially.

Step2: Push 0 into queue and mark it visited.



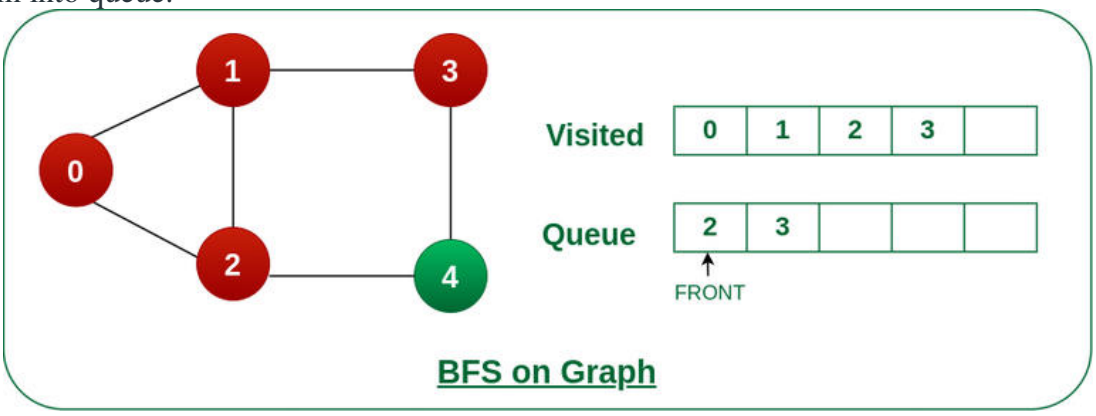
Push node 0 into queue and mark it visited.

Step 3: Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.



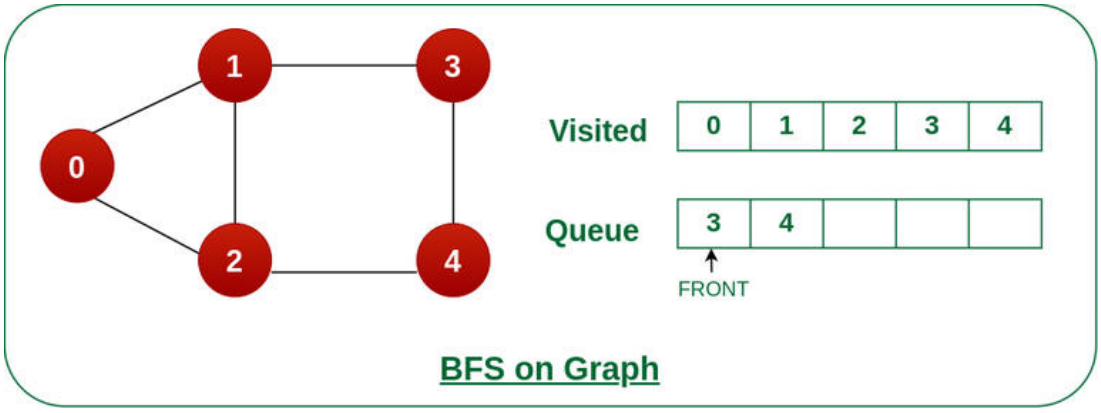
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



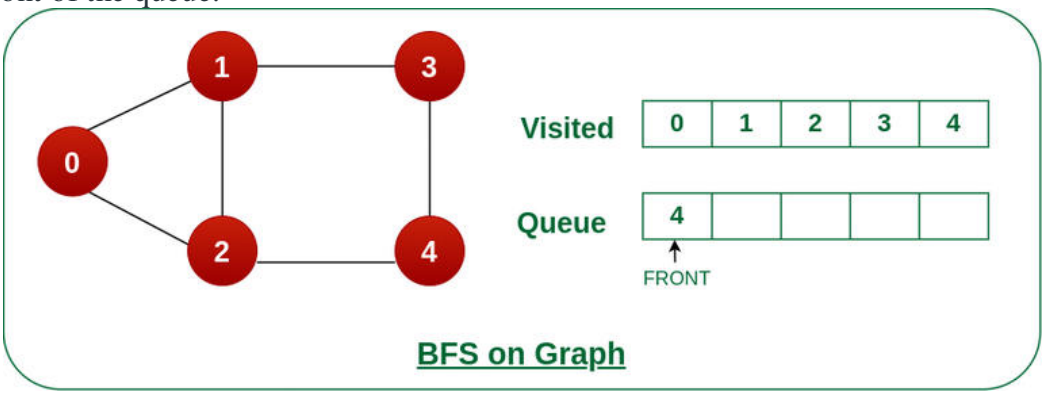
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



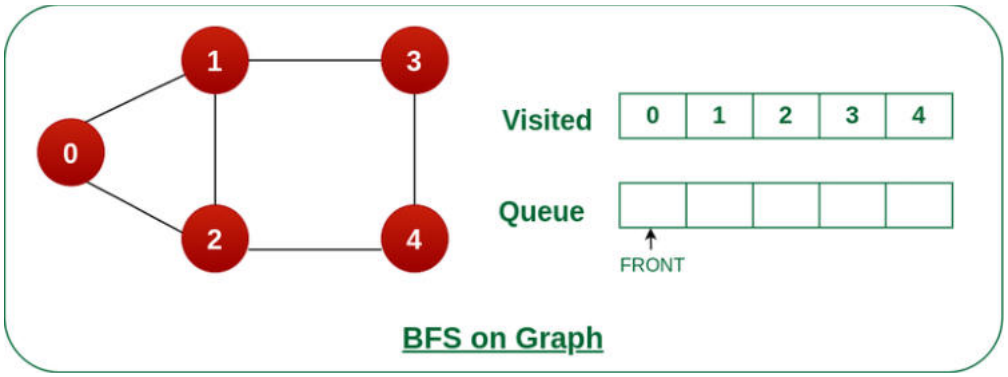
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

Comparison of BFS and DFS:

Feature	BFS	DFS
Data Structure	Queue	Stack (or recursion)
Traversal Order	Layer by layer (level-order)	Deep into one path, then backtrack
Shortest Path	Yes, in unweighted graphs	No (does not guarantee shortest path)
Space Complexity	O(V) (stores all vertices at a level)	O(V) (depth of recursion or stack)
Typical Applications	Shortest path, level-order traversal	Cycle detection, topological sorting

Both algorithms are widely used in various graph-related problems, each with its own strengths depending on the application.

Minimum Cost Spanning Tree

A **Minimum Cost Spanning Tree** (MCST) is a subset of the edges of a connected, weighted graph that connects all the vertices without any cycles and with the minimum possible total edge weight. It's essential in optimization problems, particularly in network design, where the goal is to connect all points (vertices) using the least amount of resources (minimum edge weights).

There are two main algorithms to find a minimum cost spanning tree: **Kruskal's Algorithm** and **Prim's Algorithm**.

Algorithms:

Kruskal's Algorithm:

- Sort edges by weight.
- Add the smallest edge if it doesn't form a cycle.
- Continue until all vertices are connected.
- Best for **sparse graphs**.

Prim's Algorithm:

- Start from any vertex.
- Add the smallest edge that connects a new vertex.
- Repeat until all vertices are connected.
- Best for **dense graphs**.

Comparison:

- **Kruskal's**: Works edge by edge.
- **Prim's**: Expands from a single vertex.
- Both have time complexity around **O(E log E)** or **O(E log V)**.

Applications:

- Network design (e.g., laying cables or pipelines).
- Cluster analysis.
- Approximation in optimization problems.

Graphs Applications:

- **Social Networks**: Represent people and their connections.
- **Navigation/Maps**: Find shortest paths between locations.
- **Web Search**: Rank web pages through links.
- **Transportation Networks**: Optimize routes for flights, trains, etc.
- **Computer Networks**: Route data between devices.
- **Recommendation Systems**: Suggest movies, products, etc.
- **Circuit Design**: Connect electrical components efficiently.
- **Biology**: Study protein or gene interactions.
- **Project Scheduling**: Manage task dependencies in projects.
- **Game Theory**: Analyze strategies in games like chess.

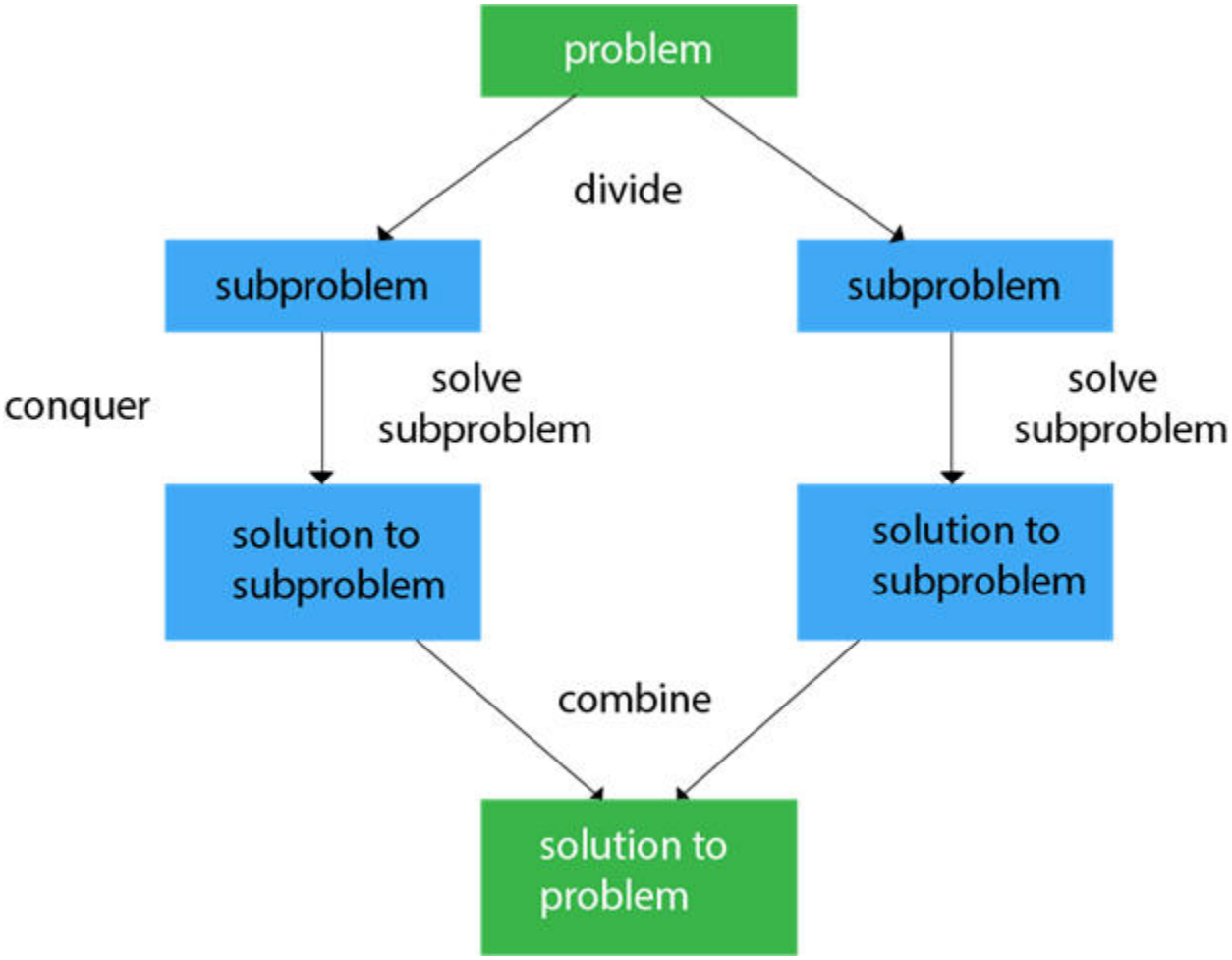
Divide and Conquer

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 subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



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

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

Advertisement

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two 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.

Applications of Divide and Conquer Approach:

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

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.
5. **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.
6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got

published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

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

Quick Sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Algorithm:

```
QUICKSORT (array A, int m, int n)
    1 if (n > m)
    2 then
    3 i ← a random index from [m,n]
    4 swap A [i] with A[m]
    5 o ← PARTITION (A, m, n)
    6 QUICKSORT (A, m, o - 1)
        7 QUICKSORT (A, o + 1, n)
```

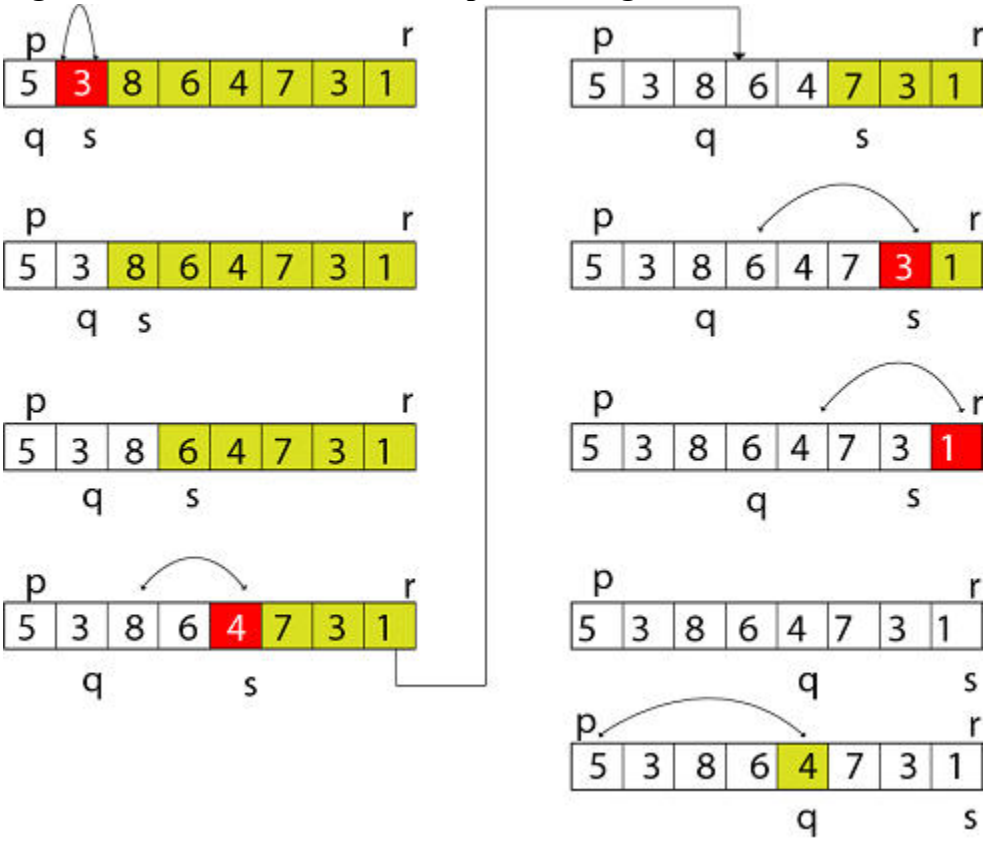
Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

```
PARTITION (array A, int m, int n)
 1 x ← A[m]
```

2 o ← m
3 **for** p ← m + 1 to n
4 **do if** (A[p] < x)
5 then o ← o + 1
6 swap A[o] with A[p]
7 swap A[m] with A[o]
8 **return** o

Figure: shows the execution trace partition algorithm



Example of Quick Sort:

44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 *33 11 55 77 90 40 60 99* ***44*** *88*

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 ***44*** *77 90 40 60 99* ***55*** *88*

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 ***40*** *77 90* ***44*** *60 99 55 88*

Swap with 77:
22 33 11 40 ***44*** *90 77 60 99 55 88*

Now, the element on the right side and left side are greater than and smaller than **44** respectively.
Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 **Sublist2**

And these sublists are sorted under the same process as above done.
These two sorted sublists side by side.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**

First sorted list

88 77 60 **55** **90** 99

Sublist3 **Sublist4**

55 77 60 **88** 90 99

Sorted

55 **77** **60**

55 60 77

Sorted

Merging Sublists:

11	**22**	**33**	**40**	**44**	**55**	**60**	**77**	**88**	**90**	**99**

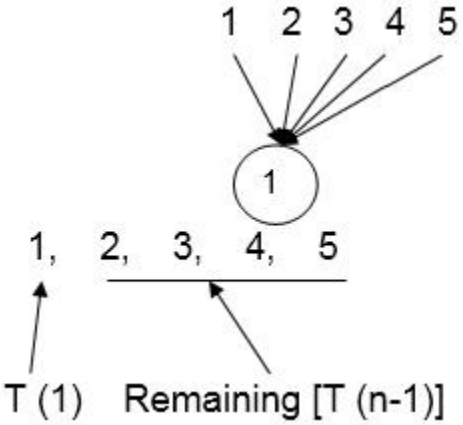
SORTED LISTS

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.
Equation:

1. T (n) =T(1)+T(n-1)+n

T (1) is time taken by pivot element.
T (n-1) is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of itself (every element)
If we compare first element pivot with other, then there will be 5 comparisons.
It means there will be n comparisons if there are n items.



Relational Formula for Worst Case:

T (n) =T (1) + T (n-1) + n..... (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

By putting (n-1) in place of n in equation1

Put T (n-1) in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n.....(ii)

T (n) =2T (1) +T (n-2) + (n-1) +n

T (n-2) =T (1) +T (n-3) + (n-2)

By putting (n-2) in place of n in equation1

Put T (n-2) in equation (ii)

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n.....(iii)

Note: for making T (n-4) as T (1) we will put (n-1) in place of '4' and if We put (n-1) in place of 4 then we have to put (n-2) in place of 3 and (n-3) In place of 2 and so on.

T(n)=(n-1) T(1) + T(n-(n-1))+(n-(n-2))+(n-(n-3))+(n-(n-4))+n
T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+.....n
T (n) = (n-1) T (1) +T (1) +2+3+4+.....+n+1-1

[Adding 1 and subtracting 1 for making AP series]

$T(n) = (n-1)\,T(1) + T(1) + 1+2+3+4+\ldots\ldots + n-1$

$T(n) = (n-1)\,T(1) + T(1) + \frac{n(n+1)}{2} - 1$

Stopping Condition: T (1) =0

Because at last there is only one element left and no comparison is required.

$T(n) = (n-1)(0) + 0 + \frac{n(n+1)}{2} - 1$

$T(n) = \frac{n^2+n-2}{2}$

Avoid all the terms expect higher terms n^2

$T(n) = O(n^2)$

Worst Case Complexity of Quick Sort is T (n) =O (n^2)

Randomized Quick Sort [Average Case]:

Generally, we assume the first element of the list as the pivot element. In an average Case, the number of chances to get a pivot element is equal to the number of items.

1. Let total time taken =T (n)
2. For eg: In a given list
3. p 1, p 2, p 3, p 4.....pn
4. If p 1 is the pivot list then we have 2 lists.
5. I.e. T (0) and T (n-1)
6. If p2 is the pivot list then we have 2 lists.
7. I.e. T (1) and T (n-2)
8. p 1, p 2, p 3, p 4.....pn
9. If p3 is the pivot list then we have 2 lists.
10. I.e. T (2) and T (n-3)
11. p 1, p 2, p 3, p 4.....p n

So in general if we take the **Kth** element to be the pivot element.

Then,

$T(n) = \sum_{K=1}^{n} T(K-1) + T(n-k)$

Pivot element will do n comparison and we are doing average case so,

$T(n) = n+1 + \frac{1}{n}\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)$

N comparisons — Average of n elements

So Relational Formula for Randomized Quick Sort is:

$T(n) = n+1 + \frac{1}{n}\left(\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)\right)$

$= n+1 + \frac{1}{n}(T(0)+T(1)+T(2)+\ldots T(n-1)+T(n-2)+T(n-3)+\ldots T(0))$

$= n+1 + \frac{1}{n} x2\ (T(0)+T(1)+T(2)+...T(n-2)+T(n-1))$

1. n T (n) = n (n+1) +2 (T(0)+T(1)+T(2)+...T(n-1).....eq 1

Put n=n-1 in eq 1

1. (n -1) T (n-1) = (n-1) n+2 (T(0)+T(1)+T(2)+...T(n-2).....eq2

From eq1 and eq 2

n T (n) - (n-1) T (n-1)= n(n+1)-n(n-1)+2 (T(0)+T(1)+T(2)+?T(n-2)+T(n-1))-2(T(0)+T(1)+T(2)+...T(n-2))

n T(n)- (n-1) T(n-1)= n[n+1-n+1]+2T(n-1)

n T(n)=[2+(n-1)]T(n-1)+2n

n T(n)= n+1 T(n-1)+2n

$\frac{n}{n+1}T(n) = \frac{2n}{n+1} + T(n-1)$ [Divide by n+1]

$\frac{1}{n+1}\ T(n) = \frac{2}{n+1} + \frac{T(n-1)}{n}$ [Divide by n] .....eq 3

Put n=n-1 in eq 3

$\frac{1}{n}T\ (n-1) = \frac{2}{n} + \frac{T(n-2)}{n-1}$.....eq4

Put 4 eq in 3 eq

$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1}$.....eq5

Put n=n-2 in eq 3

$\frac{T(n-2)}{n-1} = \frac{2}{n-1} + \frac{2}{n} + \frac{T(n-3)}{n-2}$.....eq6

Put 6 eq in 5 eq

$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{T(n-3)}{n-2}$.....eq7

Put n=n-3 in eq 3

$\frac{T(n-3)}{n-2} = \frac{2}{n-2} + \frac{T(n-4)}{n-3}$.....eq8

Put 8 eq in 7 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{T(n-4)}{n-3} \ldots\ldots\ldots\ldots \text{eq9}$$

2 terms of $\frac{2}{n+1} + \frac{2}{n} = T(n-2)$

3 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} = T(n-3)$

4 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} = T(n-4)$

From 3eq, 5eq, 7eq, 9 eq we get

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots + \frac{2}{3} + \frac{T(n-(n-1))}{n-(n-2)} \ldots\ldots\ldots \text{eq10}$$

From 3 eq $\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}$

Put n=1

$$\frac{T(1)}{2} = \frac{2}{2} + \frac{T(0)}{1}$$

$$\frac{T(1)}{2} = 1$$

From 10 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + \frac{T(1)}{2}$$

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + 1$$

Multiply and divide the last term by 2

$$=\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\ldots\ldots\ldots\ldots+\frac{2}{3}+2\times\frac{1}{2}$$

$$=2\left[\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\ldots\ldots+\frac{1}{n}+\frac{1}{n+1}\right]$$

$$=2\sum_{2\le k\le n+1}^{n}\frac{1}{k}=2\int_{2}^{n+1}\frac{1}{k}$$

Multiply & divide k by n

$$=2\int_{2}^{n+1}\frac{1}{\frac{kn}{n}}$$

Put $\frac{k}{n}=x$ and $\frac{1}{n}=dx$

$$\frac{T(n)}{n+1}=2\int_{2}^{n+1}\frac{1}{x}\,dx$$

$$=2\log x\int_{2}^{n+1}$$

NOTE: $\int\frac{1}{x}dx=\log x$

=2[log (n+1)-log2]

T (n) =2(n+1) [log (n+1)-log2]

Ignoring Constant we get

T (n) =n log n

T (n) =O (n log n)

Is the average case complexity of quick sort for sorting n elements.

3. Quick Sort [Best Case]: In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Method Name	Equation	Stopping Condition	Complexities
1.Quick Sort[Worst Case]	T(n)=T(n-1)+T(0)+n	T(1)=0	$T(n)=n^2$
2.Quick Sort[Average Case]	$T(n)=n+1+\frac{1}{n}(\sum_{K=1}^{n}T(k-1)+T(n-k))$		T(n)=nlogn

Merge Sort

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array **A**, such that our main concern will be to sort the subsection, which starts at index **p** and ends at index **r**, represented by **A[p..r]**.

Divide

If assumed **q** to be the central point somewhere in between **p** and **r**, then we will fragment the subarray **A[p..r]** into two arrays **A[p..q]** and **A[q+1, r]**.

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays **A[p..q]** and **A[q+1, r]**. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays **A[p..q]** and **A[q+1, r]**, after which we merge them back to form a new sorted array **[p..r]**.

Merge Sort algorithm

The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

ALGORITHM-MERGE SORT

1. If p<r
2. Then q → (p+ r)/2
3. MERGE-SORT (A, p, q)
4. MERGE-SORT (A, q+1,r)
5. MERGE (A, p, q, r)

Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

The following figure illustrates the dividing (splitting) procedure.

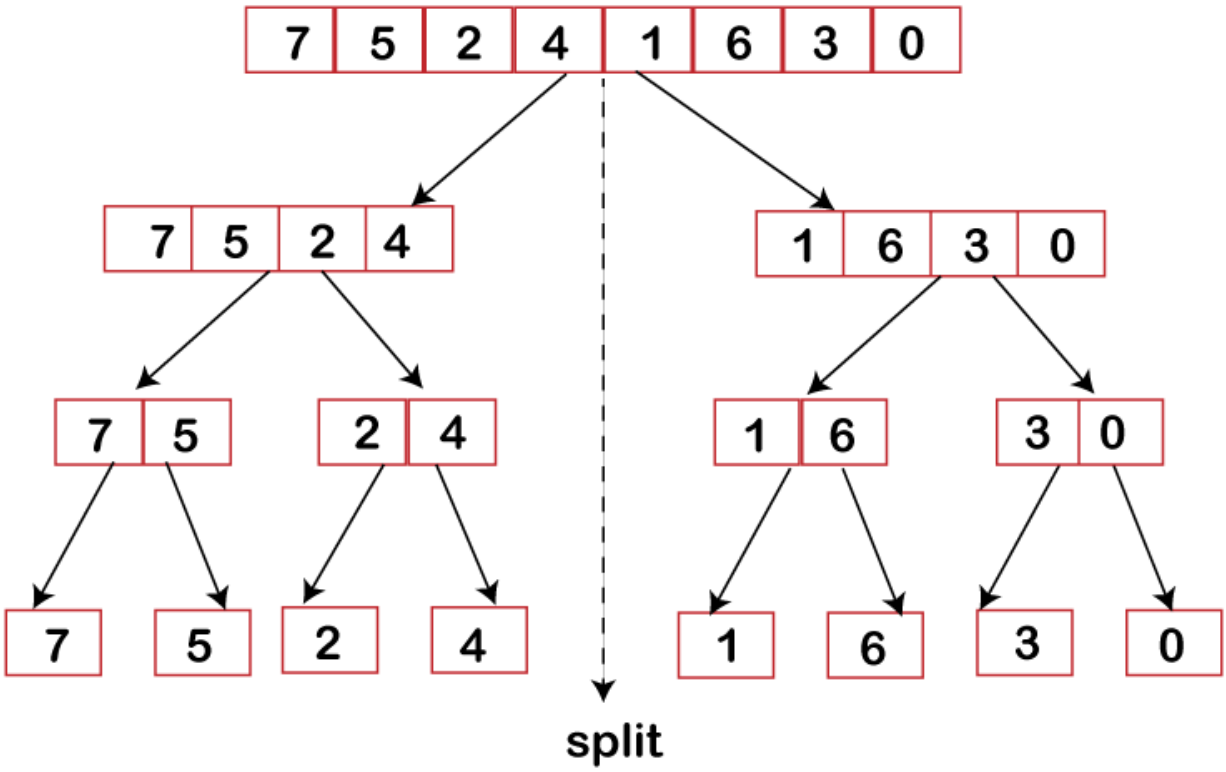


Figure 1: Merge Sort Divide Phase

FUNCTIONS: MERGE (A, p, q, r)

1. n 1 = q-p+1
2. n 2= r-q
3. create arrays [1.....n 1 + 1] and R [1.....n 2 +1]
4. for i ← 1 to n 1
5. do [i] ← A [p+ i-1]
6. for j ← 1 to n2
7. do R[j] ← A[q + j]
8. L [n 1+ 1] ← ∞
9. R[n 2+ 1] ← ∞
10. I ← 1
11. J ← 1
12. For k ← p to r
13. Do if L [i] ≤ R[j]
14. then A[k] ← L[i]
15. i ← i +1
16. else A[k] ← R[j]
17. j ← j+1

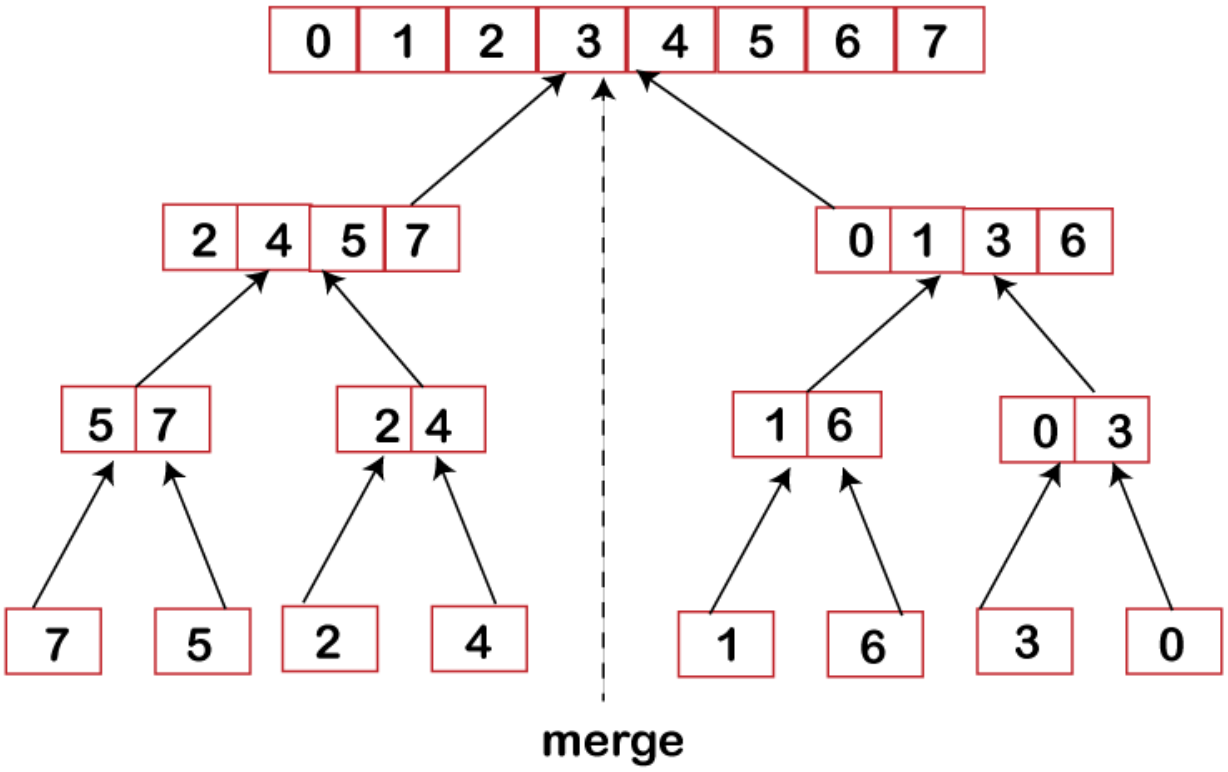


Figure 2: Merge Sort Combine Phase

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

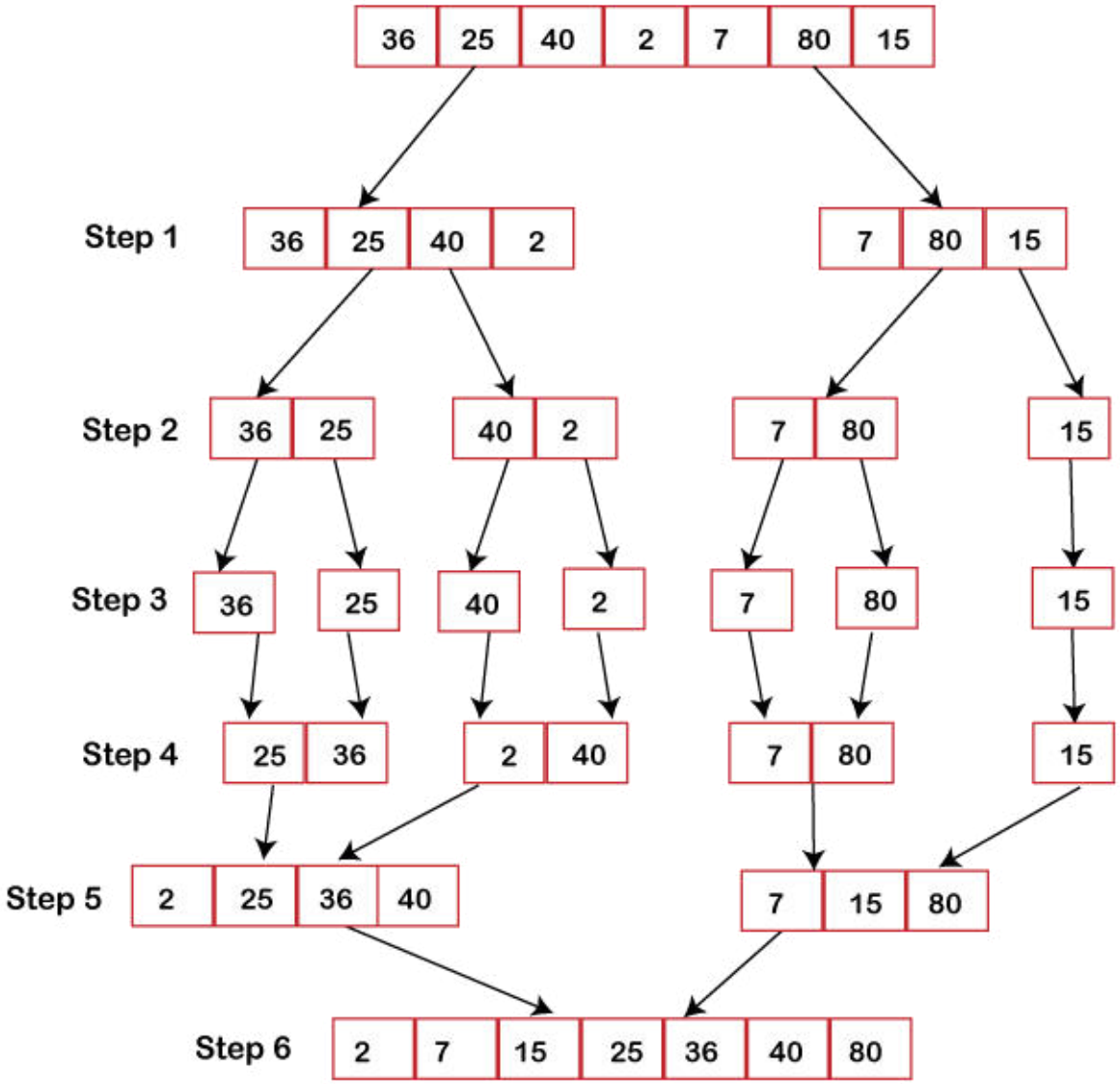
1. Did you reach the end of the array?
2. No:
3. Firstly, start with comparing the current elements of both the arrays.
4. Next, copy the smaller element into the sorted array.
5. Lastly, move the pointer of the element containing a smaller element.
6. Yes:
7. Simply copy the rest of the elements of the non-empty array

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

A= (36,25,40,2,7,80,15)

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.
Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.
Step4: Next, we will merge them back in the same way as they were broken down.
Step5: For each list, we will first compare the element and then combine them to form a new sorted list.
Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



Hence the array is sorted.

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

Note: Stopping Condition T (1) =0 because at last, there will be only 1 element left that need to be copied, and there will be no comparison.

Putting $n=\frac{n}{2}$ in place of n in .....equation 1

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$.....equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$$T(n) = 2^2\,T\left(\frac{n}{2^2}\right) + 2n$$.....equation 3

Putting $n=\frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$.....equation4

Putting 4 equation in 3 equation

$$T(n) = 2^2 \left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2^2}\right] + 2n$$

$$T(n) = 2^3 \; T\left(\frac{n}{2^3}\right) + n + 2n$$

$$T(n) = 2^3 \; T\left(\frac{n}{2^3}\right) + 3n \text{.....equation5}$$

From eq 1, eq3, eq 5.....we get

$$T(n) = 2^i \, T\left(\frac{n}{2^i}\right) + in \text{.....equation6}$$

From Stopping Condition:

$$\frac{n}{2^i} = 1 \text{ And } T\left(\frac{n}{2^i}\right)=0$$

$$n = 2^i$$

Apply log both sides:

$\log n = \log_2 i$

$\log n = i \log 2$

$$\frac{\log n}{\log 2} = i$$

$\log_2 n = i$

From 6 equation

$$T(n) = 2^i \, T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \, . \, n$$

$= T(n) = n.\log n$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

Strassen's Matrix Multiplication

Let A and B be two n×n Matrices. The product matrix C=AB is also a n×n matrix whose i, jth element is formed by taking elements in the ith row of A and jth column of B and multiplying them to get

C(i, j)=$\sum_{1\le k\le n} A(i,k)B(k,j)$

Here $1\le$ i & j $\le$ n means i and j are in between 1 and n.

To compute C(i, j) using this formula, we need n multiplications.

The divide and conquer strategy suggests another way to compute the product of two n×n matrices.

For Simplicity assume n is a power of 2 that is n=2^k

Here k→ any nonnegative integer.

If n is not power of two then enough rows and columns of zeros can be added to both A and B, so that resulting dimensions are a power of two.

Let A and B be two n×n Matrices. Imagine that A & B are each partitioned into four square sub matrices. Each sub matrix having dimensions n/2×n/2.

The product of AB can be computed by using previous formula.

If AB is product of 2×2 matrices then

$$\begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}\begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}=\begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix}$$

$C_{11}=A_{11}B_{11}+A_{12}B_{21}$
$C_{12}=A_{11}B_{12}+A_{12}B_{22}$
$C_{21}=A_{21}B_{11}+A_{22}B_{21}$
$C_{22}=A_{21}B_{12}+A_{22}B_{22}$

Here 8 multiplications and 4 additions are performed.

Note that Matrix Multiplication are more Expensive than matrix addition and subtraction.

$$T(n)=\begin{cases} b & \text{if } n\le 2; \\ 8T(n/2)+cn^2 & \text{if } n>2 \end{cases}$$

Volker strassen has discovered a way to compute the $C_{i,j}$ of above using 7 multiplications and 18 additions or subtractions.

For this first compute 7 n/2×n/2 matrices P, Q, R, S, T, U & V

P=(A11+A22)(B11+B22)
Q=(A21+A22)B11
R=A11(B12-B22)
S=A22(B21-B11)
T=(A11+A12)B22
U=(A21-A11)(B11+B12)
V=(A12-A22)(B21+B22)

C11=P+S-T+V
C12=R+T
C21=Q+S
C22=P+R-Q+U

$$T(n)=\begin{cases} b & \text{if } n\le 2; \\ 7T(n/2)+cn^2 & \text{if } n>2 \end{cases}$$

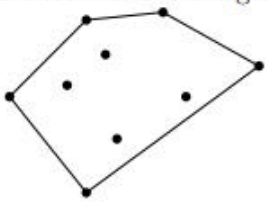
Using this recurrence relation, we get T(n)=O(nlog7).
Hence, the complexity of Strassen's matrix multiplication algorithm is O(nlog7).

Convex Hull

Chapter 1: Convex Hulls: An Example

A polygon is ***convex*** if any line segment joining two points on the boundary stays within the polygon. Equivalently, if you walk around the boundary of the polygon in counter-clockwise direction you always take left turns.

The ***convex hull*** of a set of points in the plane is the smallest convex polygon for which each point is either on the boundary or in the interior of the polygon. One might think of the points as being nails sticking out of a wooden board: then the convex hull is the shape formed by a tight rubber band that surrounds all the nails. A ***vertex*** is a corner of a polygon. For example, the highest, lowest, leftmost and rightmost points are all vertices of the convex hull. Some other characterizations are given in the exercises.



We discuss three algorithms: Graham Scan, Jarvis March and Divide & Conquer. We present the algorithms under the ***assumption*** that:

- *no 3 points are collinear (on a straight line)*

1.1 Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order.

Let x_0 be the leftmost point (which is guaranteed to be in the convex hull) and number the remaining points by angle from x_0 going counterclockwise: $x_1, x_2, \ldots, x_{n-1}$. Let $x_n = x_0$, the chosen point. Assume that no two points have the same angle from x_0.

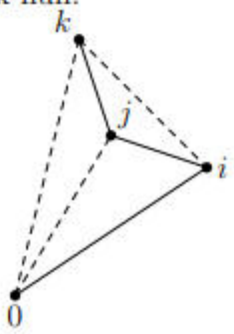
The algorithm is simple to state with a single stack:

Graham Scan

1. Sort points by angle from x_0
2. Push x_0 and x_1. Set i=2
3. While $i \leq n$ do:
   If x_i makes left turn w.r.t. top 2 items on stack
   then { push x_i; i++ }
   else { pop and discard }

To prove that the algorithm works, it suffices to argue that:

- *A discarded point is not in the convex hull.* If x_j is discarded, then for some $i < j < k$ the points $x_i \to x_j \to x_k$ form a right turn. So, x_j is inside the triangle x_0, x_i, x_k and hence is not on the convex hull.



- *What remains is convex.* This is immediate as every turn is a left turn.

The running time: Each time the while loop is executed, a point is either stacked or discarded. Since a point is looked at only once, the loop is executed at most $2n$ times. There is a constant-time subroutine for checking, given three points in order, whether the angle is a left or a right turn (Exercise). This gives an $O(n)$ time algorithm, apart from the initial sort which takes time $O(n \log n)$. (Recall that the notation $O(f(n))$, pronounced "order $f(n)$", means "asymptotically at most a constant times $f(n)$".)

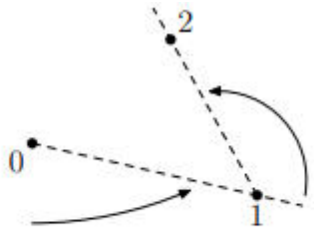
1.2 Jarvis March

This is also called the ***wrapping algorithm***. This algorithm finds the points on the convex hull ***in the order*** in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many.

Let x_0 be the leftmost point. Let x_1 be the first point counterclockwise when viewed from x_0. Then x_2 is the first point counterclockwise when viewed from x_1, and so on.

Jarvis March

$i = 0$
while not done do
    x_{i+1} = first point counterclockwise from x_i



Finding x_{i+1} takes linear time. The while loop is executed at most n times. More specifically, the while loop is executed h times where h is the number of vertices on the convex hull. So Jarvis March takes time $O(nh)$.

The best case is $h = 3$. The worst case is $h = n$, when the points are, for example, arranged on the circumference of a circle.

1.3 Divide and Conquer

Divide and Conquer is a popular technique for algorithm design. We use it here to find the convex hull. The first step is a Divide step, the second step is a Conquer step, and the third step is a Combine step.

The idea is to:

Divide and conquer

1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Part 2 is simply two ***recursive*** calls. The first point to notice is that, if a point is in the overall convex hull, then it is in the convex hull of any subset of points that contain it. (Use characterization in exercise.) So the task is: given two convex hulls find the convex hull of their union.

◇ *Combining two hulls*

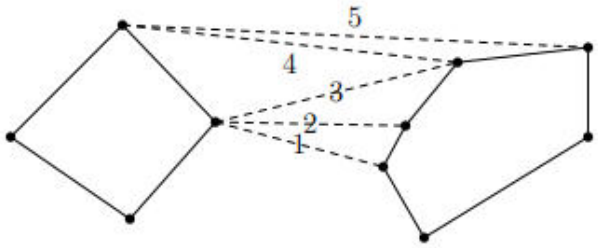
It helps to work with convex hulls that do not overlap. To ensure this, all the points are ***presorted*** from left to right. So we have a left and right half and hence a left and right convex hull.

Define a ***bridge*** as any line segment joining a vertex on the left and a vertex on the right that does not cross the side of either polygon. What we need are the ***upper*** and ***lower*** bridges. The following produces the upper bridge.

1. Start with any bridge. For example, a bridge is guaranteed if you join the rightmost vertex on the left to the leftmost vertex on the right.

2. Keeping the left end of the bridge fixed, see if the right end can be raised. That is, look at the next vertex on the right polygon going clockwise, and see whether that would be a (better) bridge. Otherwise, see if the left end can be raised while the right end remains fixed.

3. If made no progress in (2) (cannot raise either side), then stop else repeat (2).



We need to be sure that one will eventually stop. Is this obvious?

Now, we need to determine the running time of the algorithm. The key is to perform step (2) in constant time. For this it is sufficient that each vertex has a pointer to the next vertex going clockwise and going counterclockwise. Hence the choice of data structure: we store each hull using a ***doubly linked circular linked list***.

It follows that the total work done in a merge is proportional to the number of vertices. And as we shall see in the next chapter, this means that the overall algorithm takes time $O(n \log n)$.

UNIT – III

Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths.

Dynamic Programming: General Method, All pairs shortest paths, Single Source Shortest Paths – General Weights (Bellman Ford Algorithm), Optimal Binary Search Trees, 0/1 Knapsack, String Editing, Travelling Salesperson problem.

1. GENERAL METHOD

Greedy method: It is most straight forward method. It is popular for obtaining the optimized solutions.

Optimization Problem: An optimization problem is the problem of finding the best solution (optimal solution) from all the feasible solutions (practicable of possible solutions). In an optimization problem we are given a set of constraints and an optimization functions. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called optimal solution.

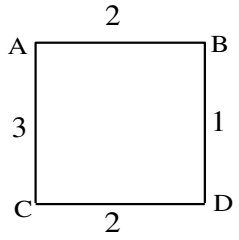
Ex: Problem: Finding a minimum spanning tree from a weighted connected directed graph G.

Constraints: Every time a minimum edge is added to the tree and adding of an edge does not form a simple circuit.

Feasible solutions: The feasible solutions are the spanning trees of the given graph G.

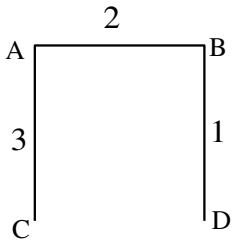
Optimal solution: An optimal solution is a spanning tree with minimum cost i.e. minimum spanning tree.

Q: Find the minimum spanning tree for the following graph.

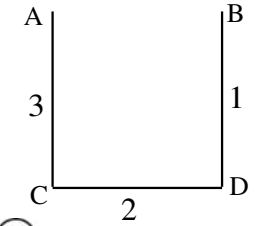


Graph G

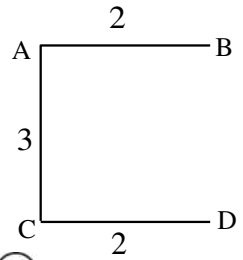
The feasible solutions are the spanning tree of the graph G. Those are



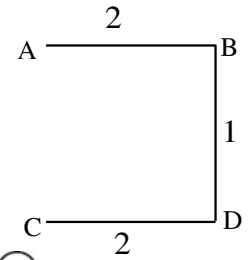
(1).Total Weights=6



(2).Total Weights=6



(3).Total Weights=7



(4).Total Weights=5

From the above spanning tree the figure 4 gives the optimal solution, because it is the spanning tree with the minimum cost i.e. it is a minimum spanning tree of the graph G.

The greedy technique suggests constructing a solution to an optimization problem hrough a sequence of steps, each expanding a partially constructed solution obtained so far until a complete solution to the problem is reached to each step, the choice made must be feasible, locally optimal and irrecoverable.

Feasible: The choice which is made has to be satisfying the problems constraints.

Locally optimal: The choice has to be the best local choice among all feasible choices available on that step.

Irrecoverable: The choice once made cannot be changed on sub-sequent steps of the algorithm (Greedy method).

Control Abstraction for Greedy Method:

```
Algorithm GreedyMethod (a, n)
{
      // a is an array of n inputs
      Solution: =Ø;
      for i: =0 to n do
      {
            s: = select (a);
            if (feasible (Solution, s)) then
            {
                  Solution: = union (Solution, s);
            }
            else
                  reject (); // if solution is not feasible reject it.
       }
       return solution;
  }
```

In greedy method there are three important activities.

1. A selection of solution from the given input domain is performed, i.e. *s:= select(a).*
2. The feasibility of the solution is performed, by using feasible '*(solution, s)*' and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objection function is obtained. Such a solution is called optimal solution.

Q: A child buys a candy 42 rupees and gives a 100 note to the cashier. Then the cashier wishes to return change using the fewest number of coins. Assume that the cashier has Rs.**1**, Rs. **5** and Rs. **10** coins.

This problem can be solved using the greedy method.

2. APPLICATION - JOB SEQUENCING WITH DEADLINES

This problem consists of ***n*** jobs each associated with a deadline and profit and our objective is to earn ***maximum profit***. We will earn profit only when job is completed on or before deadline. We assume that each job will take ***unit time*** to complete.

Points to remember:

- In this problem we have n jobs j1, j2, … jn, each has an associated deadlines are d1, d2, … dn and profits are p1, p2, ... pn.
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled or processed at any given time.

Example: Consider the following 5 jobs and their associated deadline and profit.

index	1	2	3	4	5
JOB	j1	j2	j3	j4	j5
DEADLINE	2	1	3	2	1
PROFIT	60	100	20	40	20

Sort the jobs according to their profit in descending order.

Note! If two or more jobs are having the same profit then sorts them as per their entry in the job list.

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

Find the maximum deadline value

Looking at the jobs we can say the max deadline value is 3. So, dmax = 3

As dmax = 3 so we will have THREE slots to keep track of free time slots. Set the time slot status to EMPTY

time slot	1	2	3
status	EMPTY	EMPTY	EMPTY

Total number of jobs is 5. So we can write n = 5.

Note!

If we look at job j2, it has a deadline 1. This means we have to complete job j2 in time slot 1 if we want to earn its profit.

Similarly, if we look at job j1 it has a deadline 2. This means we have to complete job j1 on or before time slot 2 in order to earn its profit.

Similarly, if we look at job j3 it has a deadline 3. This means we have to complete job j3 on or before time slot 3 in order to earn its profit.

Our objective is to select jobs that will give us higher profit.

time slot	1	2	3
Job	J1	J2	J4
Profit	100	60	20

Total Profit is 180

Pseudo Code:

```
for i = 1 to n do
  Set k = min(dmax, DEADLINE(i))  //where DEADLINE(i) denotes deadline of ith job
  while k >= 1 do
    if timeslot[k] is EMPTY then
      timeslot[k] = job(i)
      break
    endif
    Set k = k - 1
endwhile
endfor
```

Algorithm:

```
int JS(int d[], int j[], int 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.
   int k=1;
   for (int i=2; i<=n; i++) {
   //Consider jobs in nonincreasing
   // order of p[i]. Find position for
   // i and check feasibility of insertion.
      int r = k;
      while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
      if ((d[J[r]] <= d[i]) && (d[i] > r)) {
         // Insert i into J[].
         for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
         J[r+1] = i; k++;
      }
   }
   return (k);
}
```

Time Complexity = O (n^2)

3. APPLICATION - KNAPSACK PROBLEM

In this problem the objective is to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack. And we are also allowed to take an item in fractional part.

Points to remember:

In this problem we have a Knapsack that has a weight limit W

There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it v1, v2, ..., vn

Our objective is to maximise the benefit such that the total weight inside the knapsack is at most W. And we are also allowed to take an item in fractional part.

$$\max \sum_{0 \le i < n} p_i x_i$$

$$s.t.$$

$$\sum_{0 \le i < n} w_i x_i \le M$$

$$0 \le x_i <= 1$$

$$p_i \ge 0, w_i \ge 0, \; 0 \le i < n$$

Example: Assume that we have a knapsack with max weight capacity, W = 16. Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value

ITEM	WEIGHT	VALUE
i1	6	6
i2	10	2
i3	3	1
i4	5	8
i5	1	3
i6	3	5

Steps

1. Calculate value per weight for each item (we can call this value density)
2. Sort the items as per the value density in descending order
3. Take as much item as possible not already taken in the knapsack

Compute density = (value/weight)

ITEM	WEIGHT	VALUE	DENSITY
i1	6	6	1.000
i2	10	2	0.200
i3	3	1	0.333
i4	5	8	1.600
i5	1	3	3.000
i6	3	5	1.667

Sort the items as per density in descending order

ITEM	WEIGHT	VALUE	DENSITY
i5	1	3	3.000

i6	3	5	1.667
i4	5	8	1.600
i1	6	6	1.000
i3	3	1	0.333
i2	10	2	0.200

Now we will pick items such that our benefit is maximum and total weight of the selected items is at most W.

Our objective is to fill the knapsack with items to get maximum benefit without crossing the weight limit W = 16.

How to fill Knapsack Table?

is WEIGHT(i) + TOTAL WEIGHT <= W
if its YES
then we take the whole item

How to find the Benefit?

If an item value is 10 and weight is 5
And if you are taking it completely
Then,
benefit = (weight taken) x (total value of the item / total weight of the item)

weight taken = 5 (as we are taking the complete (full) item, no fraction)
total value of the item = 10
total weight of the item = 5

So, benefit = 5 x (10/5) = 10
On the other hand if you are taking say, 1/2 of the item
Then,
weight taken = 5 x (1/2) = 5/2 (as we are taking 1/2 item)
So, benefit = (weight taken) x (total value of the item / total weight of the item)
= (5/2) x (10/5)
= 5

Values after calculation

ITEM	WEIGHT	VALUE	TOTAL WEIGHT	TOTAL BENEFIT
i5	1	3	1.000	3.000
i6	3	5	4.000	8.000

i4	5	8	9.000	16.000
i1	6	6	15.000	22.000
i3	1	0.333	16.000	22.333

So, total weight in the knapsack = 16 and total value inside it = 22.333336

$n=3, m=20, (p_1,p_2,p_3)=(25,24,15), (w_1,w_2,w_3)=(18,15,10)$

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Algorithm:

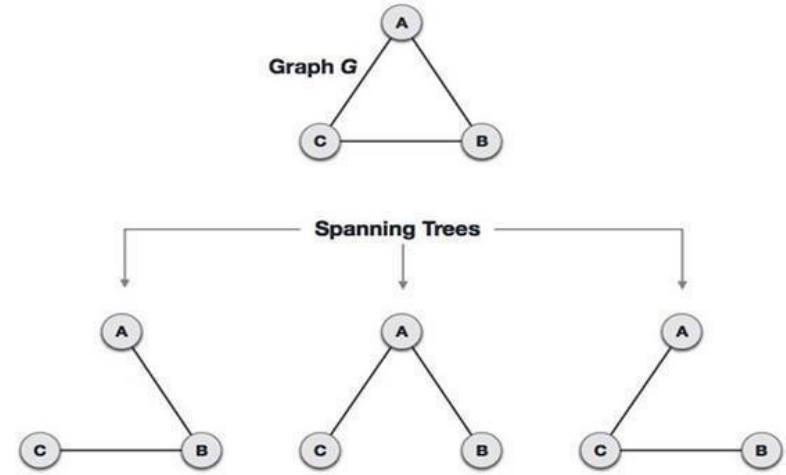
```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
   for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
   float U = m;
   for (i=1; i<=n; i++) {
      if (w[i] > U) break;
      x[i] = 1.0;
      U -= w[i];
   }
   if (i <= n) x[i] = U/w[i];
}
```

Time Complexity = O (n^2)

4. APPLICATION - MINIMUM SPANNING TREE

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

Note: Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum nn-2 number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

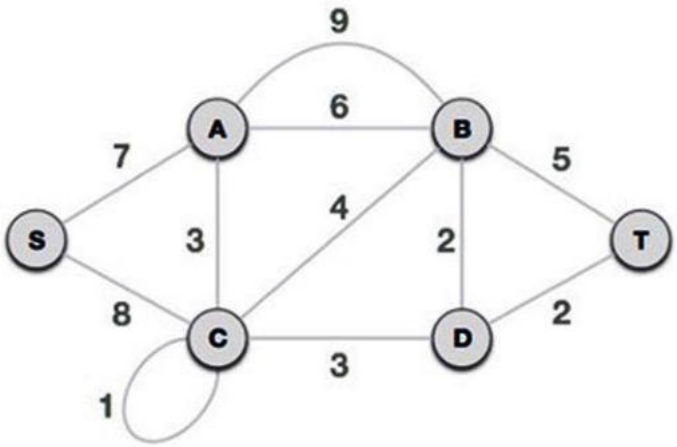
We shall learn about two most important spanning tree algorithms(greedy algorithms):

1. Kruskal's Algorithm
2. Prim's Algorithm

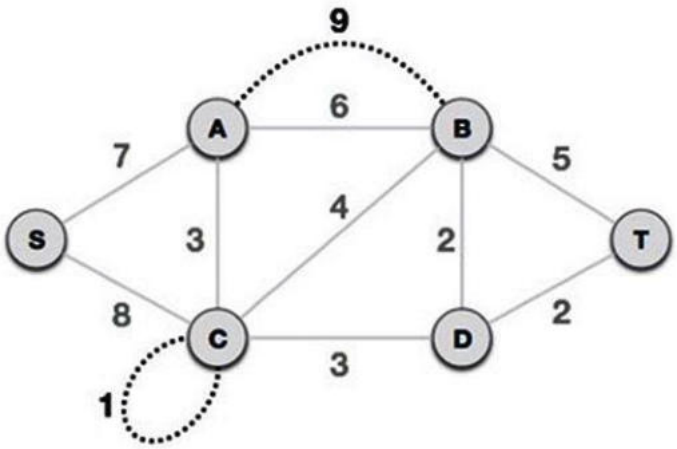
i. Kruskal's Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

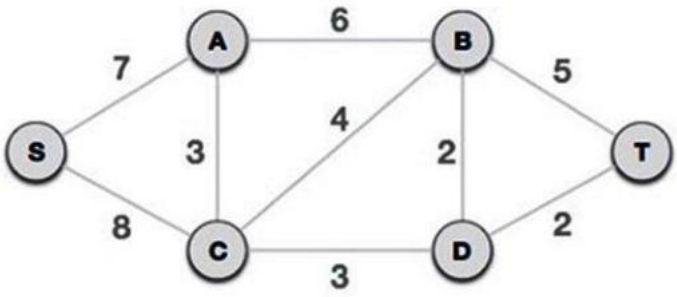
To understand Kruskal's algorithm let us consider the following example:



Step 1 - Remove all loops and Parallel Edges
Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



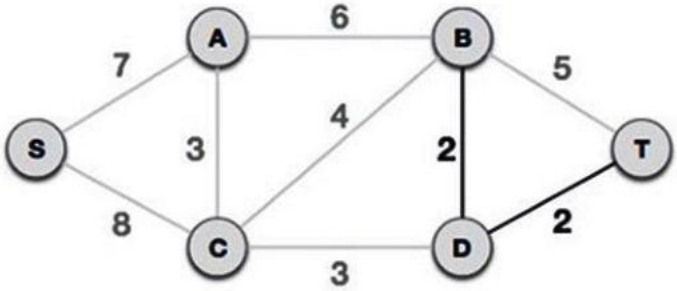
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

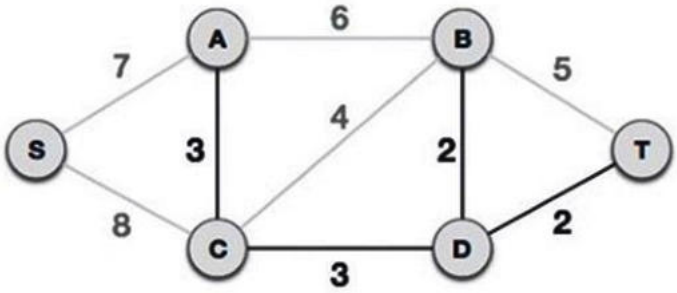
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

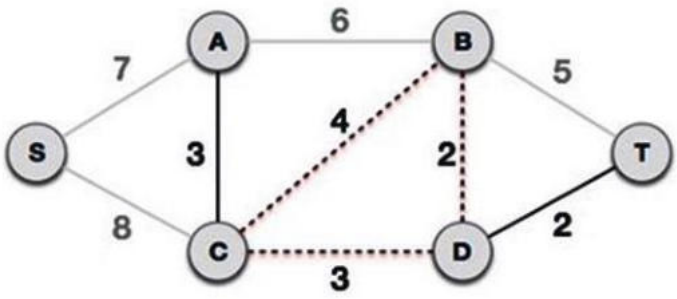


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

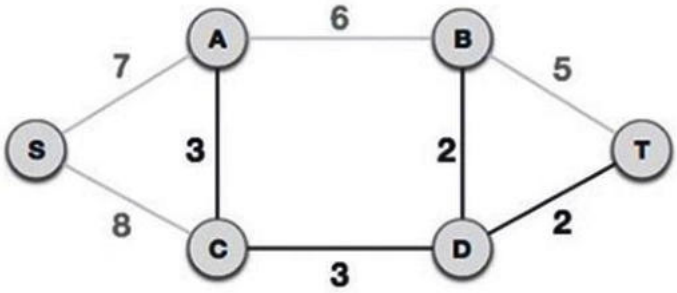
Next cost is 3, and associated edges are A,C and C,D. We add them again −



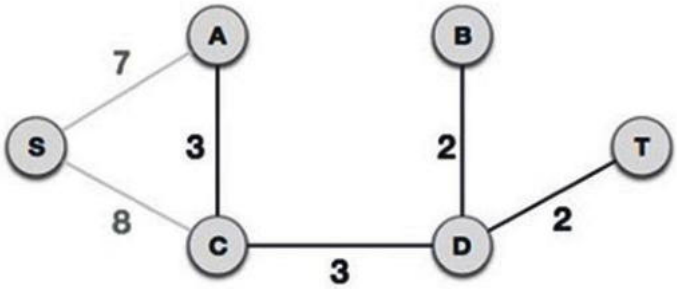
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



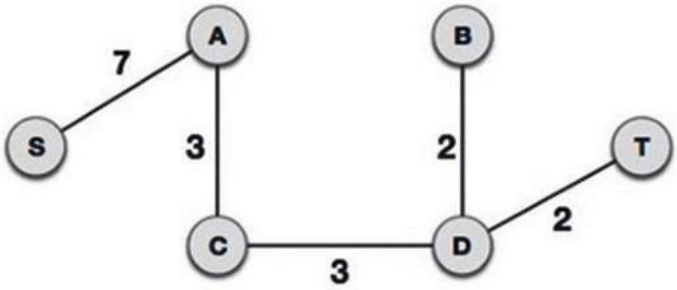
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

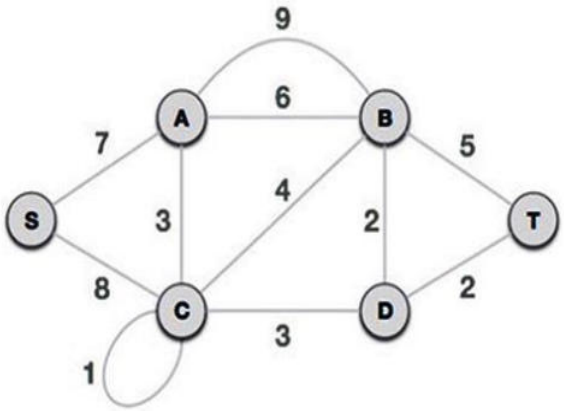
```
float Kruskal(int E[][SIZE], float cost[][SIZE], int n, int t[][2])
{
   int parent[SIZE];
   construct a heap out of the edge costs using Heapify;
   for (int i=1; i<=n; i++) parent[i] = -1;
   // Each vertex is in a different set.
   i = 0; float mincost = 0.0;
   while ((i < n-1) && (heap not empty)) {
      delete a minimum cost edge (u,v) from the heap
      and reheapify using Adjust;
      int j = Find(u); int k = Find(v);
      if (j != k) {
            i++;
            t[i][1] = u; y[i][2] = v;
            mincost += cost[u][v];
            Union(j, k);
      }
   }
    if ( i != n-1) cout << "No spanning tree" << endl;
    else return(mincost);
}
```

Time Complexity = O (|E| log |E|)

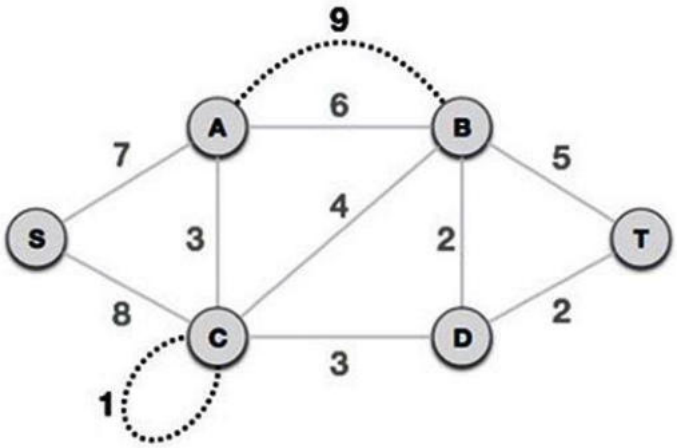
ii. Prim's Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

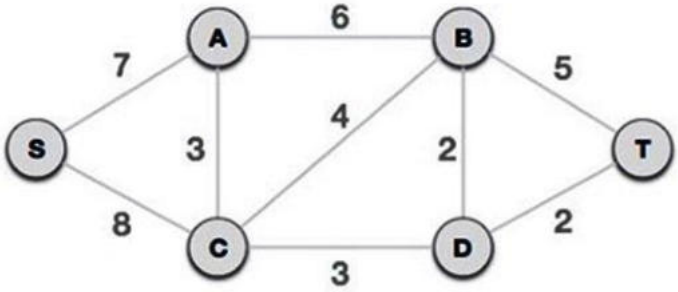
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example.



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

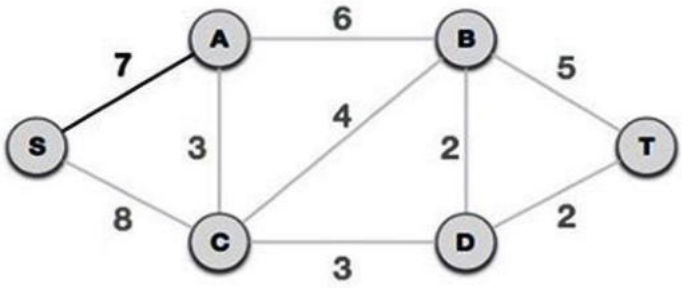


Step 2 - Choose any arbitrary node as root node

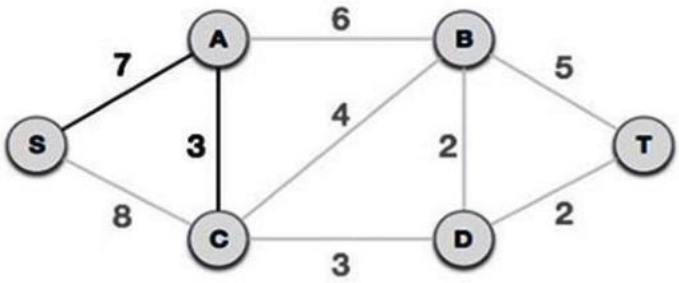
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

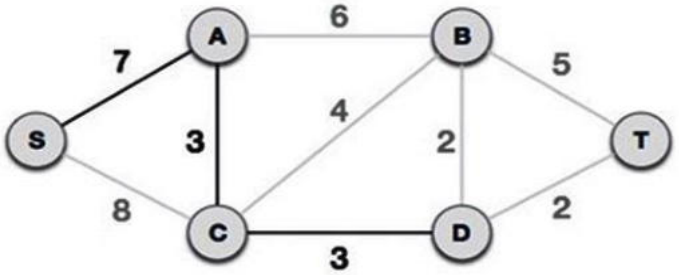
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



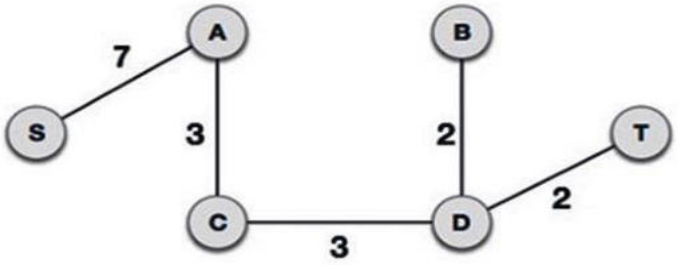
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

```
1   float Prim(int E[][SIZE], float cost[][SIZE],  int n, int t[][2])
11  {
12    int near[SIZE], j, k, L;
13    let (k,L) be an edge of minimum cost in E;
14    float mincost = cost[k][L];
15    t[1][1] = k; t[1][2] = L;
16    for (int i=1; i<=n; i++)  // Initialize near.
17      if (cost[i][L] < cost[i][k]) near[i] = L;
18      else near[i] = k;
19    near[k] = near[L] = 0;
20    for (i=2; i <= n-1; i++) { // Find n-2 additional
21                               // edges for t.
22      let j be an index such that near[j]!=0 and
23      cost[j][near[j]] is minimum;
24      t[i][1] = j; t[i][2] = near[j];
25      mincost = mincost + cost[j][near[j]];
26      near[j]=0;
27      for (k=1; k<=n; k++) // Update near[].
28        if ((near[k]!=0) &&
29            (cost[k][near[k]]>cost[k][j]))
30          near[k] = j;
31    }
32     return(mincost);
33  }
```

Time Complexity = O (n^2)

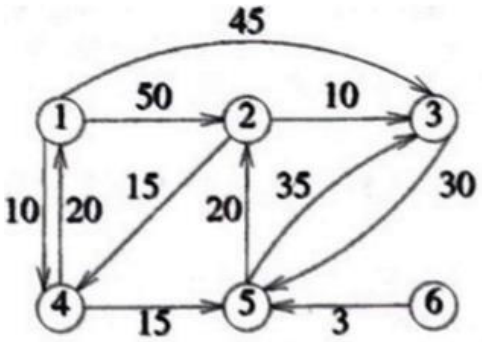
5. APPLICATION - SINGLE SOURCE SHORTEST PATH PROBLEM

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It also used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Example:



(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

Algorithm:

```
ShortestPaths(int v, float cost[][SIZE], float dist[], int n)
{
    int u; bool S[SIZE];
    for (int i=1; i<= n; i++) { // Initialize S.
        S[i] = false; dist[i] = cost[v][i];
    }
    S[v]=true; dist[v]=0.0; // Put v in S.
    for (int num = 2; num < n; num++) {
     // 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 (int w=1; w<=n; w++) //Update distances.
            if (( S[w]=false) && (dist[w] > dist[u] + cost[u][w]))
                dist[w] = dist[u] + cost[u][w];
        }
}
```

Time Complexity = O (n^2)

GREEDY APPROACH	DIVIDE AND CONQUER
1.Many decisions and sequences areguaranteed and all the overlapping subinstancesare considered.	1.Divide the given problem into many subproblems.Find the individual solutions andcombine them to get the solution for themain problem
2. Follows Bottom-up technique	2. Follows top down technique
3.Split the input at every possible pointsrather than at a particular point	3.Split the input only at specific points (midpoint), each problem is independent.
4. Sub problems are dependent on the main Problem	4. Sub problems are independent on the main Problem
5. Time taken by this approach is not that much efficient when compared with DAC.	5. Time taken by this approach efficient when compared with GA.
6.Space requirement is less when compared DAC approach.	6.Space requirement is very much high when compared GA approach.

UNIT III

Dynamic Programming

The general method, multistage graphs, All pairs-shortest paths, optimal Binary search trees, 0/1 knapsack, The traveling salesperson problem.

……

1Q) The general method

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use Memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

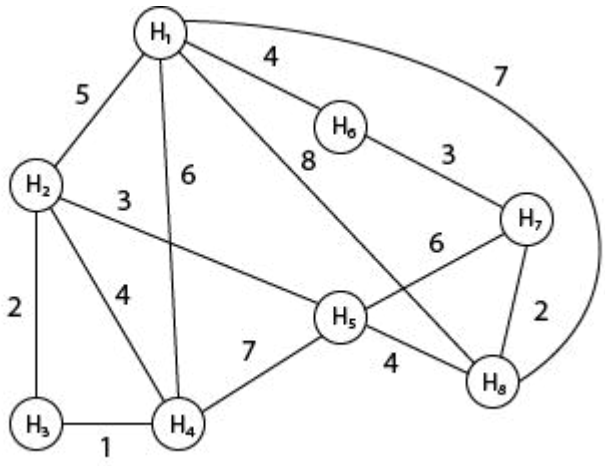
Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

2Q) The traveling salesperson problem (TSP)

The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

4 3 2 4 3 2 1 6

$H_1 \rightarrow H_6 \rightarrow H_7 \rightarrow H_8 \rightarrow H_5 \rightarrow H_2 \rightarrow H_3 \rightarrow H_4 \rightarrow {}_{H1}$.

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Time Complexity

The recursive equation is

$$M(i,j) = \min_{i \le k < j} [M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j].$$

Using the above recurrence relation, we can write a dynamic programming-based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than O(n!) but still exponential. The space required is also exponential. So this approach is also infeasible even for a slightly higher number of vertices. We will soon be discussing approximate algorithms for the traveling salesman problem.

The dynamic programming approach breaks the problem into $2^n n$ subproblems. Each subproblem takes n time resulting in a time complexity of $O(2^n n^2)$. Here n refers to the number of cities needed to be travelled too.

3Q) 0/1 knapsack Problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.

For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Example of 0/1 knapsack problem.

Consider the problem having weights and profits are:

Weights: {2,3,4,5}

Profits: {1,2,5,6}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

How this problem can be solved by using the Dynamic programming approach?

First,

we create a matrix shown as below:

		0	1	2	3	4	5	6	7	8
P	W	0								
1	2	1								
2	3	2								
5	4	3								
6	5	4								

In the above matrix, columns represent the weight, i.e., 8. The rows represent the profits and weights of items. Here we have not taken the weight 8 directly, problem is divided into sub-problems, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8. The solution of the sub-problems would be saved in the cells and answer to the problem would be stored in the final cell. First, we write the weights in the ascending order and profits according to their weights

The first row and the first column would be 0 as there is no item for w=0

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0							
2	3	2	0							
5	4	3	0							
6	5	4	0							

Now, the weight of the first object is '2' and it can be filled only when the weight is 2. Fill the respective cell with corresponding profit.

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0							
5	4	3	0							
6	5	4	0							

While, we are selecting second object we will also consider first object also . The weight of the second object is 3 and it will be filled only bag weight is 3. Fill the corresponding cell with respective profit. Now the total weight including first and 2nd objects are 2+3=5 (profit is 1+2=3). It will be filled when the weight of the bag is 5.

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0							
6	5	4	0							

While selecting third object we also consider first two objects also. The weight of 3rd object is 4 and it profit is 5.

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5				
6	5	4	0							

Now the combinations for filling remaining positions are

Object1+object3 = 2+4=6(W) = 1+5=6(P)

Object3+Object2 = 4+3=7(W)=2+5=7(P)

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5	5	6	7	7
6	5	4	0	1	2	5	6	6	7	8

Similarly, we will fill the last row also like previous step.

As we can observe in the above table that 8 is the maximum profit among all the entries.

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5	5	6	7	7
6	5	4	0	1	2	5	6	6	6	8

➔ We will select the fourth object (x_4) (with profit 6), the reaming profit is 8-6 =2

Now check weather 2 is there in the 3rd object, as we can see there is no two(x_3), look at 2nd row, yes two is there(x_3) check weather two is there in 2nd row or not, if it is there don't select it(x_3). Move to the 2nd object and check for two, if there also check above row for two, if not present include it in the solution.

		0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5	5	6	7	7
6	5	4	0	1	2	5	6	6	6	8

The reaming profit is zero. The object included in the bag are X2 and X4

X1	X2	X3	X4
0	1	0	1

Time Complexity of the above approach is O(N*W).

Space Complexity of the above approach is O(W).

```
PW = record {float p; float w; }

1   Algorithm DKnap(p, w, x, n, m)
2   {
3       // pair[ ] is an array of PW's.
4       b[0] := 1; pair[1].p := pair[1].w := 0.0; // S^0
5       t := 1; h := 1; // Start and end of S^0
6       b[1] := next := 2; // Next free spot in pair[ ]
7       for i := 1 to n - 1 do
8       { // Generate S^i.
9           k := t;
10          u := Largest(pair, w, t, h, i, m);
11          for j := t to u do
12          { // Generate S_1^{i-1} and merge.
13              pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
14                  // (pp, ww) is the next element in S_1^{i-1}.
15              while ((k ≤ h) and (pair[k].w ≤ ww)) do
16              {
17                  pair[next].p := pair[k].p;
18                  pair[next].w := pair[k].w;
19                  next := next + 1; k := k + 1;
20              }
21              if ((k ≤ h) and (pair[k].w = ww)) then
22              {
23                  if pp < pair[k].p then pp := pair[k].p;
24                  k := k + 1;
25              }
26              if pp > pair[next - 1].p then
27              {
28                  pair[next].p := pp; pair[next].w := ww;
29                  next := next + 1 ;
30              }
31              while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p))
32                  do k := k + 1;
33          }
34          // Merge in remaining terms from S^{i-1}.
35          while (k ≤ h) do
36          {
37              pair[next].p := pair[k].p; pair[next].w := pair[k].w;
38              next := next + 1; k := k + 1;
39          }
40          // Initialize for S^{i+1}.
41          t := h + 1; h := next - 1; b[i + 1] := next;
42      }
43      TraceBack(p, w, pair, x, m, n);
44  }
```

Algorithm 5.7 Algorithm for 0/1 knapsack problem

4) optimal Binary search trees

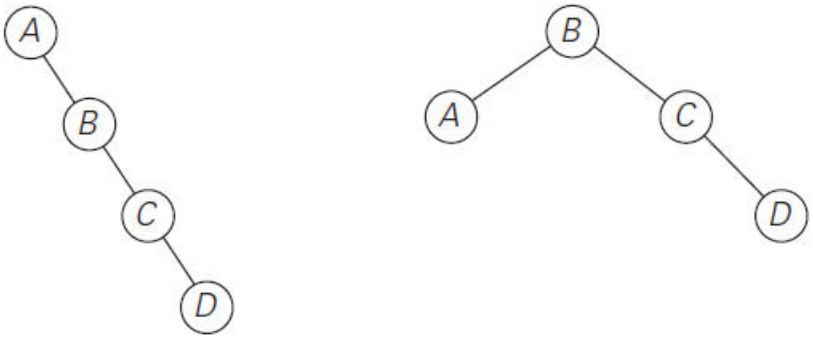
4.1 Explain about Optimal Binary Search Tree with Successful and Unsuccessful search probabilities with suitable example

Answer:

OBST is a binary search tree which provides the smallest possible search time (or expected search) for a given sequence of accesses (or access probabilities).

The search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

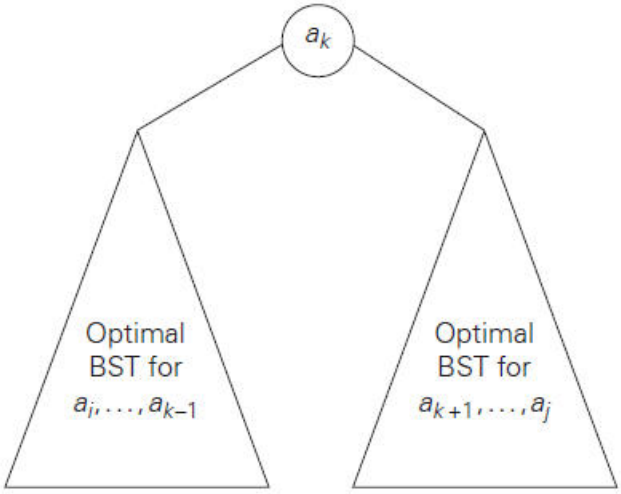
Example:



Two out of 14 possible binary search trees with keys A, B, C, and D.

For our tiny example, we could find the optimal tree by generating all 14 binary search trees with 4 keys. As a general algorithm, this exhaustive-search approach is unrealistic: the total number of binary search trees with n keys is equal to the nth **Catalan number**,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$



Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^{j}.

Given a set of identifiers {a1,a2,..,an}. Suppose we need to construct a binary search tree and p(i) be the probability with which we search for a_i then:

If a binary search tree represents n identifiers, then there will be exactly n internal nodes and n+1 external nodes. Every node internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

If a successful search terminates at an internal node at level l, then l comparison is needed. Hence the expected cost contribution from the internal node for a_i is p(i)*level(a_i).

The identifiers not in the binary search tree can be partitioned into n+1 equivalence classes Ei, $0 \le i \le n$. If the failure node for Ei is at level l, then only l-1 comparison are needed.

Let q(i) be the probability that the identifier x being searched for is in E_i, then clearly $\sum_{i=1}^{n} Pi + \sum_{i=0}^{n} qi = 1$, and the cost contribution for the failure node for E_i is q(i)*(level (E_i) - 1).

There fore, the cost of the optimal binary search tree is:

$$\sum_{i=1}^{n} Pi * level(ai) + \sum_{i=0}^{n} qi * (level(Ei) - 1)$$

```
Algorithm OBST(p, q, n)
// Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities
// p[i], 1 <= i <= n, and q[i], 0 <= i <= n, this algorithm computes
// the cost c[i,j] of optimal binary search trees t_ij for identifiers
// a_{i+1}, ..., a_j. It also computes r[i,j], the root of t_ij.
// w[i,j] is the weight of t_ij.
{
    for i := 0 to n - 1 do
    {
        // Initialize.
        w[i,i] := q[i]; r[i,i] := 0; c[i,i] := 0.0;
        // Optimal trees with one node
        w[i,i+1] := q[i] + q[i+1] + p[i+1];
        r[i,i+1] := i + 1;
        c[i,i+1] := q[i] + q[i+1] + p[i+1];
    }
    w[n,n] := q[n]; r[n,n] := 0; c[n,n] := 0.0;
    for m := 2 to n do  // Find optimal trees with m nodes.
        for i := 0 to n - m do
        {
            j := i + m;
            w[i,j] := w[i,j-1] + p[j] + q[j];
            // Solve 5.12 using Knuth's result.
            k := Find(c, r, i, j);
                // A value of l in the range r[i,j-1] <= l
                // <= r[i+1,j] that minimizes c[i,l-1] + c[l,j];
            c[i,j] := w[i,j] + c[i,k-1] + c[k,j];
            r[i,j] := k;
        }
    write (c[0,n], w[0,n], r[0,n]);
}

Algorithm Find(c, r, i, j)
{
    min := ∞;
    for m := r[i,j-1] to r[i+1,j] do
        if (c[i,m-1] + c[m,j]) < min then
        {
            min := c[i,m-1] + c[m,j]; l := m;
        }
    return l;
}
```

Time complexity: The computing time for above algorithm is $O(n^2)$. To construct obst from r[i,j] is O(n). So total time to construct obst is $O(n^3)$.

Space complexity = $O(n^2)$

Eg. construct optimal binary search tree for the below data

$(a_1, a_2, a_3, a_4) = (\text{end}, \text{goto}, \text{print}, \text{stop})$

$P(a_1) = \frac{1}{20}$ $P(a_2) = \frac{1}{5}$ $P(a_3) = \frac{1}{10}$ $P(4) = \frac{1}{20}$

$q(0) = \frac{1}{5}$ $q(1) = \frac{1}{10}$ $q(2) = \frac{1}{5}$ $q(3) = \frac{1}{20}$ $P(4) = \frac{1}{20}$

Given data set is $(a_1, a_2, a_3, a_4) = (\text{end}, \text{goto}, \text{print}, \text{stop})$

For convenience, we multiply p's & q's with 20.

Then we have,

$(P_1, P_2, P_3, P_4) = (1, 4, 2, 1)$

$(q_0, q_1, q_2, q_3, q_4) = (4, 2, 4, 1, 1)$

We Know,

$w_{i,j}$ - wt. of the tree (optimal binary search tree)

$$w_{ij} = p_j + q_j w_{i,j-1} \quad \text{if } i<j$$
$$= q_i \quad \text{if } i=j \qquad \text{—(1)}$$

$c(i,j)$ = cost of OBST, then

$$c(i,j) = \min_{i<K\le j} \{ c(i,K-1) + c(K,j) + w(i,j) \} \qquad \text{—(2)}$$
$$c(i,i) = 0 \quad 0 \le i \le n$$

Let $r(i,j)$ - be K value for which a_K is the root of OBST.

$r(i,j)$ = value of K that minimizes

$r(i,i) = 0 \quad 0 \le i \le n.$

For the given instance, $n = 4$

The values $w(i,j)$ $c(i,j)$ $r(i,j)$ are calculated as follows & the values were shown in the following table.

$w_{00} = 4$	$w_{11} = 2$	$w_{22} = 4$	$w_{33} = 1$	$w_{44} = 1$
$c_{00} = 0$	$c_{11} = 0$	$c_{22} = 0$	$c_{33} = 0$	$c_{44} = 0$
$r_{00} = 0$	$r_{11} = 0$	$r_{22} = 0$	$r_{33} = 0$	$r_{44} = 0$
$w_{01} = 7$	$w_{12} = 10$	$w_{23} = 7$	$w_{34} = 3$	
$c_{01} = 7$	$c_{12} = 10$	$c_{23} = 7$	$c_{34} = 3$	
$r_{01} = 1$	$r_{12} = 2$	$r_{23} = 3$	$r_{34} = 4$	
$w_{02} = 15$	$w_{13} = 13$	$w_{24} = 9$		
$c_{02} = 22$	$c_{13} = 20$	$c_{24} = 12$		
$r_{02} = 2$	$r_{13} = 2$	$r_{24} = 3$		
$w_{03} = 18$	$w_{14} = 15$			
$c_{03} = 32$	$c_{14} = 27$			
$r_{03} = 2$	$r_{14} = 2$			
$w_{04} = 20$				
$c_{04} = 39$				
$r_{04} = 2$				

Computations:

From (1) we have $w_{ii} = q(i)$ for $i = 0$ to 4

$\therefore w_{00} = q_0 = 4$, $w_{11} = q_1 = 2$, $w_{22} = q_2 = 4$ $w_{33} = q_3 = 1$

$w_{44} = q_4 = 1$.

From (2) we have $c_{ii} = 0$ & $r_{ii} = 0$ for $i = 0$ to 4.

- $w(0,1) = p_1 + q_1 + w(0,0) = 1+2+4 = 7$

$c(0,1) \Rightarrow i=0, j=1.$

$k=1 \Rightarrow c_{01} = \min\{c_{00}+c_{11}\} + w_{01}$

$= (0+0)+7 = 7.$

$r_{01} = 1.$

- $w_{12} = p_2 + q_2 + w_{11} = 4+4+2 = 10.$

$c_{12} \overset{k=2}{=} \min\{c_{11}+c_{22}\} + w_{12} = (0+0)+10 = 10.$

$r_{12} = 2$

- $w_{23} = p_3 + q_3 + w_{22} = 2+1+4 = 7$

c_{24} : $k=3.$

$c_{24} = \min\{c_{22}+c_{33}\} + w_{23} = (0+0)+7 = 7$

$r_{23} = 3$

- $w_{34} = p_4 + q_4 + w_{33} = 1+1+1 = 3$

c_{34} : $k=4.$

$c_{34} = \min\{c_{33}+c_{44}\} + w_{34} = (0+0)+3 = 3$

$r_{34} = 4.$

- $w_{02} = p_2 + q_2 + w_{01} = 4+4+7 = 15$

c_{02} : $k = 1, 2$

$c_{02} = \min\{\underset{k=1}{c_{00}+c_{12}}, \underset{k=2}{c_{01}+c_{22}}\} + w_{02}$

$= \min\{0+10, 7+0\} + w_{02}$

$= 7+15 = 22$

$r_{02} = 2$

• $\omega_{13} = P_3 + q_3 + \omega_{12} = 2 + 1 + 10 = 13$

c_{13} : $k = 2, 3.$

$$c_{13} = \min\{\underbrace{c_{11} + c_{23}}_{k=2}, \underbrace{c_{12} + c_{33}}_{k=3}\} + \omega_{13}$$
$$= \min\{0+7, 10+0\} + 13$$
$$= 7 + 13 = 20.$$

$r_{13} = 2.$

• $\omega_{24} = P_4 + q_4 + \omega_{23} = 1 + 1 + 7 = 9$

c_{24} : $k = 3, 4$

$$c_{24} = \min\{\underbrace{c_{22} + c_{34}}_{k=3}, \underbrace{c_{23} + c_{44}}_{k=4}\} + \omega_{24}$$
$$= \min\{0+3, 7+0\} + 9 = 3 + 9 = 12$$

$r_{24} = 3.$

• $\omega_{03} = P_3 + q_3 + \omega_{02} = 2 + 1 + 15 = 18$

c_{03} : $k = 1, 2, 3.$

$$c_{03} = \min\{\underbrace{c_{00} + c_{13}}_{k=1}, \underbrace{c_{01} + c_{23}}_{k=2}, \underbrace{c_{02} + c_{33}}_{k=3}\} + \omega_{03}$$
$$= \min\{0+20, 7+7, 22+0\} + 18 = 14 + 18 = 32$$

$r_{03} = 2$

• $\omega_{14} = P_4 + q_4 + \omega_{13} = 1 + 1 + 13 = 15$

c_{14} : $k = 2, 3, 4$

$$c_{14} = \min\{\underbrace{c_{11} + c_{24}}_{k=2}, \underbrace{c_{12} + c_{34}}_{k=3}, \underbrace{c_{13} + c_{44}}_{k=4}\} + \omega_{14}$$
$$= \min\{0+12, 10+3, 20+0\} + 15 = 27.$$

$r_{14} = 2$

$w_{04} = p_4 + q_4 + w_{03} = 1 + 1 + 18 = 20$

c_{04} : $k = 1, 2, 3, 4.$

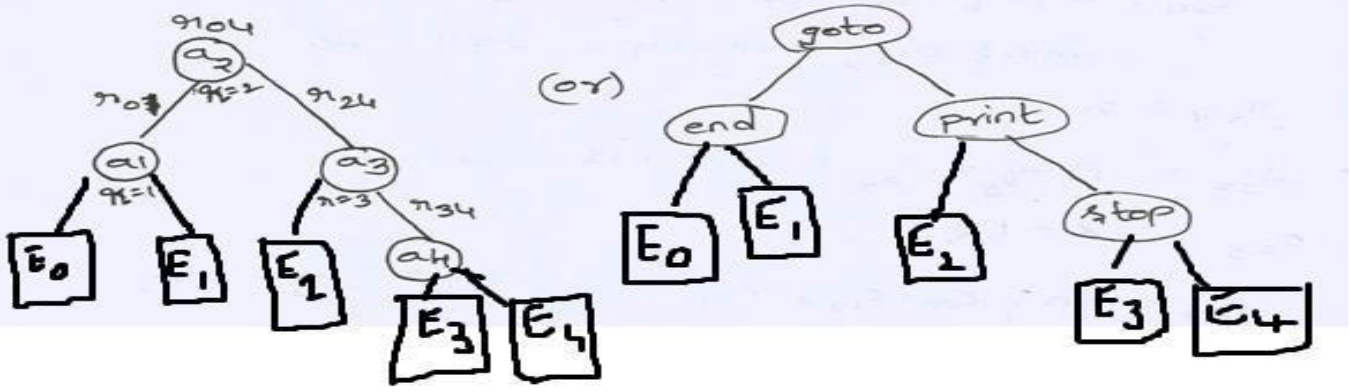
$c_{04} = \min\{ \underset{k=1}{c_{00} + c_{14}}, \underset{k=2}{c_{01} + c_{24}}, \underset{k=3}{c_{02} + c_{34}}, \underset{k=4}{c_{03} + c_{44}} \} + w_{04}$

$= \min\{0+27, 7+12, 22+3, 32+0\} + 20$

$= 19 + 20 = 39.$

$r_{04} = 2,$

∴ From above table, the cost of optimal binary search tree is $c_{04} = 39.$



Successful Search cost of the tree = 1(2) + 4(1) + 2(2) + 1(3) = 13.

Unsuccessful search cost of the tree = 4(3-1) + 2(3-1) + 4(3-1) + 1(4-1) + 1(4-1) = 26

So, total cost of the tree = 13+26 =39.

4.2 Explain about Optimal Binary Search Tree with Successful search probabilities with suitable example.

OBST is a binary search tree which provides the smallest possible search time (or expected search) for a given sequence of accesses (or access probabilities).

The search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

• Given a set of identifiers {a1,a2,..,an}. Suppose we need to construct a binary search tree and p(i) be the probability with which we search for ai then:

If a successful search terminates at an internal node at level l, then l comparison is needed. Hence the expected cost contribution from the internal node for ai is p(i)*level(ai).

• Therefore, the cost of the optimal binary search tree is:

$$\sum_{i=1}^{n} \text{Pi} * \text{level(ai)}$$

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do    //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

Time Complexity: The computing time for above algorithm is $O(n^2)$. To construct obst from r[i,j] is O(n). So total time to construct obst is $O(n^3)$.

Space complexity: $O(n^2)$

Eg. construct an optimal binary search tree for the following keys with the probabilities as

Keys	A	B	C	D	E
Prob.	0.25	0.2	0.05	0.2	0.3

For constructing OBST we have the following recurrence:

$$c(i,j) = \min_{i \le k \le j} \{c(i,k-1) + c(k+1,j)\} + \sum_{s=i}^{j} P_s \quad , 1 \le i \le j \le n \qquad \text{---(1)}$$

We assume in above formula that,

$$c(i, i-1) = 0 \quad , \quad 1 \le i \le n+1$$

that are no. of comparisions in empty tree

and from above (1),

$$c(i, i) = P_i \quad , \quad 1 \le i \le n$$

for one node binary search tree a_i.

The initial tables look like:

main table

	0	1	2	3	4	5
1	0	0.25				
2		0	0.2			
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

root table

	0	1	2	3	4	5
1		1				
2			2			
3				3		
4					4	
5						5

$$c(1,2) = \min \begin{cases} K=1 \Rightarrow c(1,0) + c(2,2) + \sum_{s=1}^{2} P_s = 0 + 0.2 + 0.25 + 0.2 = 0.65 \\ K=2 \Rightarrow c(1,1) + c(3,2) + \sum_{s=1}^{2} P_s = 0.25 + 0 + 0.25 + 0.2 = 0.7 \end{cases}$$

$\therefore c(1,2) = 0.65$ & $r_{12} = 1$

$$c(2,3) = \min \begin{cases} K=2 \Rightarrow c(2,1) + c(3,3) + \sum_{s=1}^{3} P_s = 0 + 0.05 + 0.2 + 0.05 = 0.05 + 0.25 = 0.3 \\ K=3 \Rightarrow c(2,2) + c(4,3) + \sum_{s=1}^{3} P_s = 0.2 + 0 + 0.25 = 0.45 \end{cases}$$

$\therefore c(2,3) = 0.3$, $r_{23} = 2$.

$$c(3,4) = \min \begin{cases} K=3 \Rightarrow c(3,3) + c(4,4) + \sum_{s=3}^{4} P_s = 0.05 + 0.2 + 0.05 + 0.2 = 0.25 + 0.2 = 0.45 \\ K=4 \Rightarrow c(3,3) + c(5,4) + \sum_{s=3}^{4} P_s = 0.05 + 0 + 0.25 = 0.3 \end{cases}$$

$\therefore c(3,4) = 0.3$, $r_{34} = 4$

$$c(4,5) = \min \begin{cases} K=4 \Rightarrow c(4,3) + c(5,5) + \sum_{s=4}^{5} P_s = 0 + 0.3 + 0.2 + 0.3 = 0.3 + 0.5 = 0.8 \\ K=5 \Rightarrow c(4,4) + c(6,5) + \sum_{s=4}^{5} P_s = 0.2 + 0 + 0.5 = 0.7 \end{cases}$$

$\therefore c(4,5) = 0.7$, $r_{45} = 5$.

$$c(1,3) = \min \begin{cases} K=1 \Rightarrow c(1,0) + c(2,3) + \sum_{s=1}^{3} P_s = 0 + 0.3 + 0.25 + 0.2 + 0.05 = 0.3 + 0.5 = 0.8 \\ K=2 \Rightarrow c(1,1) + c(3,3) + \sum_{s=1}^{3} P_s = 0.25 + 0.05 + 0.5 = 0.8 \\ K=3 \Rightarrow c(1,2) + c(4,3) + \sum_{s=1}^{3} P_s = 0.65 + 0 + 0.5 = 1.15 \end{cases}$$

$\therefore c(1,3) = 0.8$, $r_{13} = 1$

$$c(2,4) = \min \begin{cases} K=2 \Rightarrow c(2,1) + c(3,4) + \sum_{s=2}^{4} P_s = 0 + 0.3 + 0.2 + 0.05 + 0.2 = 0.3 + 0.45 = 0.75 \\ K=3 \Rightarrow c(2,2) + c(4,4) + \sum_{s=2}^{4} P_s = 0.2 + 0.2 + 0.45 = 0.85 \\ K=4 \Rightarrow c(2,3) + c(5,4) + \sum_{s=2}^{4} P_s = 0.3 + 0 + 0.45 = 0.75 \end{cases}$$

$\therefore c(2,4) = 0.75$, $r_{24} = 2$.

$$c(3,5) = \min \begin{cases} k=3 \Rightarrow c(3,2) + c(4,5) + \sum_{s=i}^{j} P_s = 0+0.7+0.05+0.2+0.3 \\ \qquad\qquad = 0.7+0.55 = 1.25 \\ k=4 \Rightarrow c(3,3) + c(5,5) + \sum_{s=i}^{j} P_s = 0.05+0.3+0.55 = 0.9 \\ k=5 \Rightarrow c(3,4) + c(6,5) + \sum_{s=i}^{j} P_s = 0.3+0+0.55 = 0.85 \end{cases}$$

$\therefore c(3,5) = 0.85, \quad r_{35} = 5.$

$$c(1,4) = \min \begin{cases} k=1 \Rightarrow c(1,0) + c(2,4) + \sum_{s=i}^{j} P_s = 0+0.75+0.25+0.2+0.05+0.2 \\ \qquad\qquad = 0.75+0.7 = 1.45 \\ k=2 \Rightarrow c(1,1) + c(3,4) + \sum_{s=i}^{j} P_s = 0.25+0.3+0.7 = 1.25 \\ k=3 \Rightarrow c(1,2) + c(4,4) + \sum_{s=i}^{j} P_s = 0.65+0.2+0.7 = 1.55 \\ k=4 \Rightarrow c(1,3) + c(5,4) + \sum P_s = 0.8+0+0.7 = 1.5 \end{cases}$$

$c(1,4) = 1.25, \quad r_{14} = 2.$

$$c(2,5) = \min \begin{cases} k=2 \Rightarrow c(2,4) + c(5,5) + \sum_{s=i}^{j} P_s = \overset{0+0.85}{0.75+0} + 0.2+0.05+0.2+0.3 \\ \qquad\qquad = 0.85+0.75 = 1.6 \\ k=3 \Rightarrow c(2,2) + c(4,5) + \sum_{s=i}^{j} P_s = 0.2+0.7+0.75 = 1.65 \\ k=4 \Rightarrow c(2,3) + c(5,5) + \sum P_s = 0.3+0.3+0.75 = 1.35 \end{cases}$$

$\therefore c(2,5) = 1.35, \quad r_{25} = 4$

$$c(1,5) = \min \begin{cases} k=1 \Rightarrow c(1,0) + c(2,5) + \sum_{s=i}^{j} P_s = 0+1.35+0.25+0.2+0.05+0.2+0.3 \\ \qquad\qquad = 1.35+1 = 2.35 \\ k=2 \Rightarrow c(1,1) + c(3,5) + \sum_{s=i}^{j} P_s = 0.25+0.85+1 = 2.1 \\ k=3 \Rightarrow c(1,2) + c(4,5) + \sum P_s = 0.65+0.7+1 = 2.35 \\ k=4 \Rightarrow c(1,3) + c(5,5) + \sum P_s = 0.8+0.3+1 = 2.1 \\ k=5 \Rightarrow c(1,4) + c(6,5) + \sum P_s = 1.25+0+1 = 2.25 \end{cases}$$

$\therefore c(1,5) = 2.1, \quad r_{15} = 2$

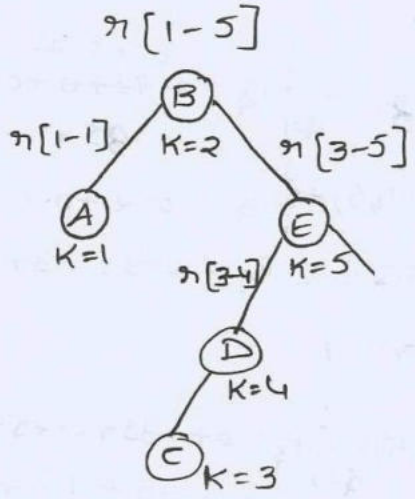
∴ The final tables are :

main table

	0	1	2	3	4	5
1	0	0.25	0.65	0.8	1.25	2.1
2		0	0.2	0.3	0.75	1.35
3			0	0.05	0.3	0.85
4				0	0.2	0.7
5					0	0.3
6						0

root table

	0	1	2	3	4	5
1		1	1	1	2	2
2			2	2	2	4
3				3	4	5
4					4	5
5						5
6						



∴ Search cost of the tree is

$2(0.25) + 1(0.2) + 4(0.05) + 3(0.2) + 2(0.3) = 2.1$

5. All pairs-shortest paths

All Pairs Shortest Path Algorithm – Introduction

All Pairs Shortest Path Algorithm is also known as the Floyd-Warshall algorithm. And this is an optimization problem that can be solved using dynamic programming.

Let G = <V, E> be a directed graph, where V is a set of vertices and E is a set of edges with nonnegative length. Find the shortest path between each pair of nodes.

L = Matrix, which gives the length of each edge

L[i, j] = 0, if i == j // Distance of node from itself is zero

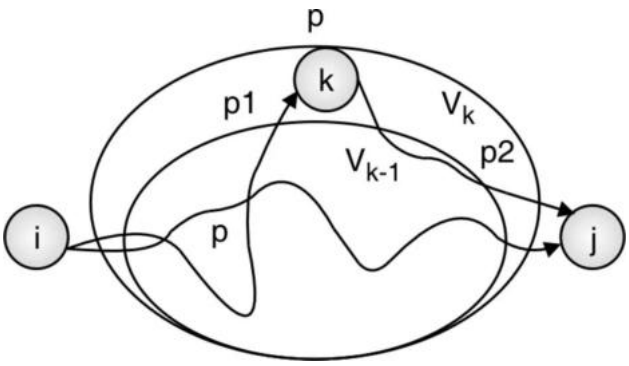
L[i, j] = ∞ , if i ≠ j and (i, j) ∉ E

L[i, j] = w (i, j), if i ≠ j and (i, j) ∈ E // w(i, j) is the weight of the edge (i, j)

Principle of optimality:

If k is the node on the shortest path from i to j, then the path from i to k and k to j, must also be shortest.

In the following figure, the optimal path from i to j is either p or summation of p_1 and p_2.



While considering k^{th} vertex as intermediate vertex, there are two possibilities:

- If k is not part of shortest path from i to j, we keep the distance D[i, j] as it is.
- If k is part of shortest path from i to j, update distance D[i, j] as D[i, k] + D[k, j].

Optimal sub structure of the problem is given as :

$D^k [i, j] = \min\{ D^{k-1} [i, j], D^{k-1} [i, k] + D^{k-1} [k, j] \}$

D^k = Distance matrix after k^{th} iteration

Algorithm for All Pairs Shortest Path

This approach is also known as the **Floyd-warshall** shortest path algorithm. The algorithm for all pair shortest path (APSP) problem is described below

Algorithm FLOYD_APSP (L)

// L is the matrix of size n × n representing original graph

// D is the distance matrix

D ← L

for k ← 1 to n **do**

  for i ← 1 to n **do**

    for j ← 1 to n **do**

$D[i, j]^k \leftarrow \min (D[i, j]^{k-1}, D[i, k]^{k-1} + D[k, j]^{k-1})$

end

end

end

return D

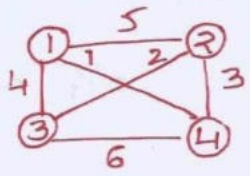
Complexity analysis of All Pairs Shortest Path Algorithm

It is very simple to derive the complexity of all pairs' shortest path problem from the above algorithm. It uses three nested loops. The innermost loop has only one statement. The complexity of that statement is Q(1).

The running time of the algorithm is computed as:

$$T(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \Theta(1) = \sum_{k=1}^{n} \sum_{i=1}^{n} n = \sum_{k=1}^{n} n^2 = O(n^3)$$

Eg. Find the shortest distance from each vertex to other vertices in the below graph.



The All Pairs shortest-path problem is to determine a matrix A such that $A(i,j)$ is the length of a shortest path i, j.

From floyd's alg. we have the recurrence relation :

$$A(i,j) = \min\left\{\min_{1\le k\le n}\{A^{k-1}(i,k)+A^{k-1}(k,j)\}, cost(i,j)\right\}$$

$$\Rightarrow A^0(i,j) = cost(i,j),\quad 1\le i\le n,\quad 1\le j\le n$$

$$\boxed{A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\},\quad k\ge 1}$$

The cost matrix for given graph is given by

$$A^0 = cost(i,j) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 6 \\ 1 & 3 & 6 & 0 \end{bmatrix} \end{array}$$

where $cost(i,j) = \begin{cases} 0 & \text{if } i=j \\ \text{edge cost}(i,j) & \text{if } i\ne j \ \&\ (i,j)\in E \\ \infty & \text{if } i\ne j \ \&\ (i,j)\notin E \end{cases}$

and E is set of edges in graph 'G'.

When computing $A^{(1)}$,

- 1st row & 1st column can be copied from $A^{(0)}$ to $A^{(1)}$ as they remain constant.
- all diagonal elements will be 0's always.

K = 1

$a^1_{23} = \min\{a^0_{23}, (a^0_{21} + a^0_{13})\}$

i = 2, j = 3

$= \min\{2, (5+4)\} = 2$

$a^1_{24} = \min\{a^0_{24}, (a^0_{21} + a^0_{14})\} = \min\{3, (5+1)\} = 3$

$a^1_{32} = \min\{a^0_{32}, (a^0_{31} + a^0_{12})\} = \min\{2, (4+5)\} = 2$

$a^1_{34} = \min\{a^0_{34}, (a^0_{31} + a^0_{14})\} = \min\{6, (4+1)\} = 5$

$a^1_{42} = \min\{a^0_{42}, (a^0_{41} + a^0_{12})\} = \min\{3, (1+5)\} = 3$

$a^1_{43} = \min\{a^0_{43}, (a^0_{41} + a^0_{13})\} = \min\{6, (1+4)\} = 5$

$$\therefore A^1 = \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & \underline{5} \\ 1 & 3 & \underline{5} & 0 \end{bmatrix}$$

11ly $$A^2 = \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

11ly $$A^3 = \begin{bmatrix} 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 0 & \underline{4} & 4 & 1 \\ \underline{4} & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

6. multistage graphs

1. Explain the methodology of Dynamic programming. Mention the applications of Dynamic programming.

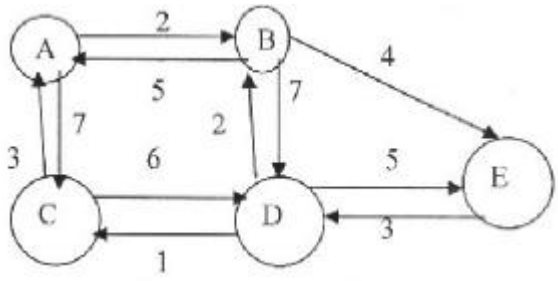
2. Find the all pairs shortest path solution for the graph represented by below adjacency matrix:

$$\begin{bmatrix} \infty & 6 & 5 & 4 \\ 3 & \infty & 2 & 6 \\ 18 & 6 & \infty & 7 \\ 8 & 12 & 10 & \infty \end{bmatrix}$$

3. Write and explain an algorithm to compute the all pairs shortest path using dynamic programming and prove that it is optimal.

4. What is All – Pair Shortest Path problem (APSP)? Discuss the Floyd's APSP algorithm and discuss the analysis of this algorithm.

5. Find the shortest path between all pairs of nodes in the following graph.



6. Write a function to compute lengths of shortest paths between all pairs of nodes for the given adjacency matrix.

$$\begin{pmatrix} 0 & 6 & 13 \\ 8 & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$$

7. Solve the following instance of 0/1 KNAPSACK problem using Dynamic programming n = 3, (W1, W2, W3) = (2, 3, 4), (P1, P2, P3) = (1, 2, 5), and m = 6.

8. Define merging and purging rules in 0/1 knapsack problem and explain with an example.

9. Describe the Dynamic 0/1 Knapsack problem. Find an optimal solution for the dynamic programming 0/1 knapsack instance for n=3, m=6, profits are (p1, p2, p3) = (1, 2, 5), weights are (w1, w2, w3)=(2, 3, 4).

10. What is principles of optimality? Explain how travelling sales person problem uses the dynamic programming technique with example?

11. Construct an optimal travelling sales person tour using Dynamic Programming For the given data

$$\begin{pmatrix} 0 & 10 & 9 & 3 \\ 5 & 0 & 6 & 2 \\ 9 & 6 & 0 & 7 \\ 7 & 3 & 5 & 0 \end{pmatrix}$$

12.Discuss the time and space complexity of Dynamic Programming traveling sales person algorithm

UNIT IV
Backtracking: General method, Applications- n-queue problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles.

BACKTRACKING

General Method:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, , x_n) where each $x_i \in S$, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x_1, , x_n). Form a solution and check at every step
if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance Iof problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i's must relate to each other.

- For 8-queens problem:

  Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7, 8}.

  The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can beon the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Terminology:

Problem state is each node in the depth first search tree.

Solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

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 node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.
All solutions to the 8-queens problem can be represented as 8-tuples (x_1, ... , x_8), where x_i is the column of the i^{th} row where the i^{th} queen is placed.

The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.
This realization reduces the size of the solution space from 8^8 tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or diagonal:
Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diag 45 conflict: Two queens i and j are on the same 45^0 diagonal if:

  $$i - j = k - l.$$

  This implies, $j - l = i - k$
- Diag 135 conflict:

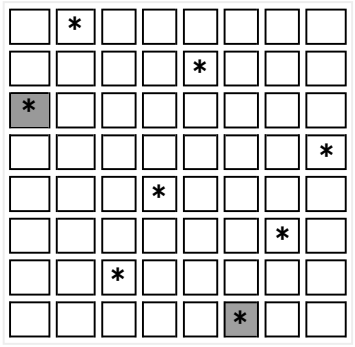
  $i + j = k + l$. This implies, $j - l = k - i$

Therefore, two queens lie on the same diagonal if and only if:

$$|j - l| = |i - k|$$

Where, j be the column of object in row i for the i^{th} queen and l be the column of object in row 'k' for the k^{th} queen.

To check the diagonal clashes, let us take the following tile configuration:



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

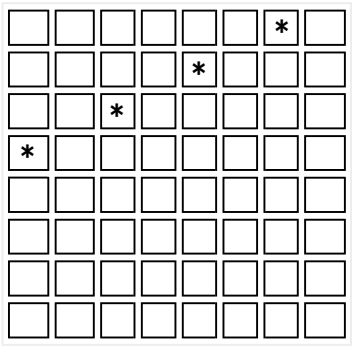
Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

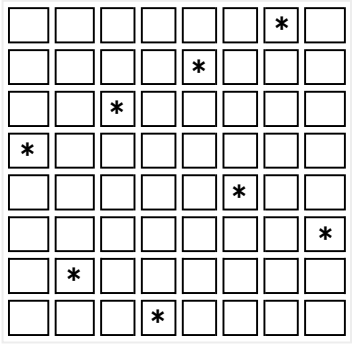
Step 3:

If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

1	2	3	4	5	6	7	8	Remarks
7	5	3	1					
7	5	3	1*	2*				|j - I| = |1 – 2| = 1 |i – k | = |4 – 5| = 1
7	5	3	1	4				
7*	5	3	1	4	2*			|j - I| = |7 – 2| = 5 |i – k | = |1 – 6| = 5
7	5	3*	1	4	6*			|j - I| = |3 – 6| = 3 |i – k | = |3 – 6| = 3
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		|j - I| = |4 – 2| = 2 |i – k | = |5 – 7| = 2
7	5	3	1	4*	8	6*		|j - I| = |4 – 6| = 2 |i – k | = |5 – 7| = 2
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*
7	5	3	1	6				
7*	5	3	1	6	2*			|j - I| = |1 – 2| = 1 |i – k | = |7 – 6| = 1
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	|j - I| = |3 – 8| = 5 |i – k| =|3 – 8| = 5
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

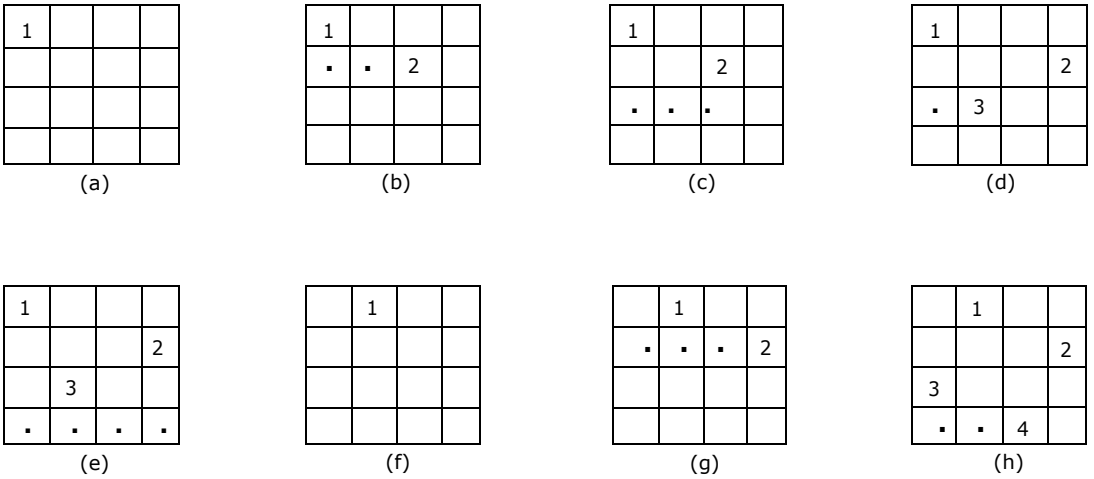
* indicates conflicting queens.

On a chessboard, the **solution** will look like:



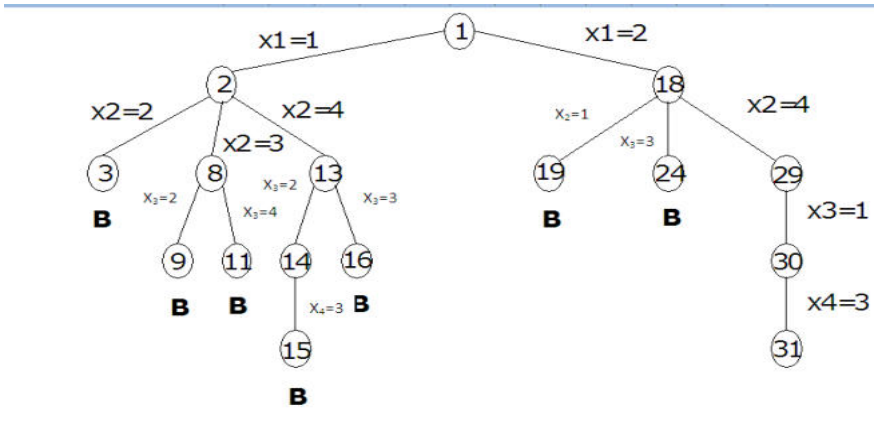
4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:



The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] − i) = Abs(j − k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

Complexity Analysis:

$$1 + n + n^2 + n^3 + \ldots\ldots\ldots\ldots\ldots\ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which n = 8, the state space tree contains:

$$\frac{8^{8+1}-1}{8-1} = 19,173,961 \text{ nodes}$$

Sum of Subsets:

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, $1 \le k \le n$.

Explicit constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \le j \le n\}$.

Implicit constraints:

- No two x_i can be the same.
- The sum of the corresponding w_i's be m.
- $x_i < x_{i+1}$, $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

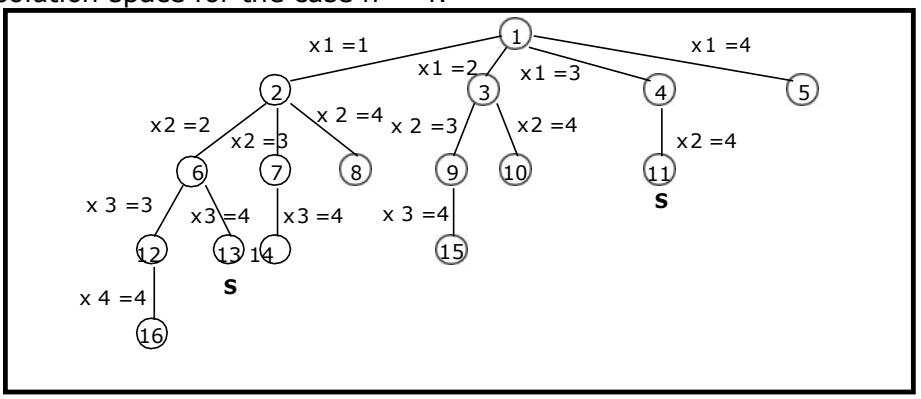
A better formulation of the problem is where the solution subset is represented by an n-tuple (x_1,..... , x_n) such that $x_i \in \{0, 1\}$.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2^n distincttuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are(11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.



A possible solution space organisation for the sum of the subsets problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path

corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left mot sub-tree defines all subsets containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, and so on.

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

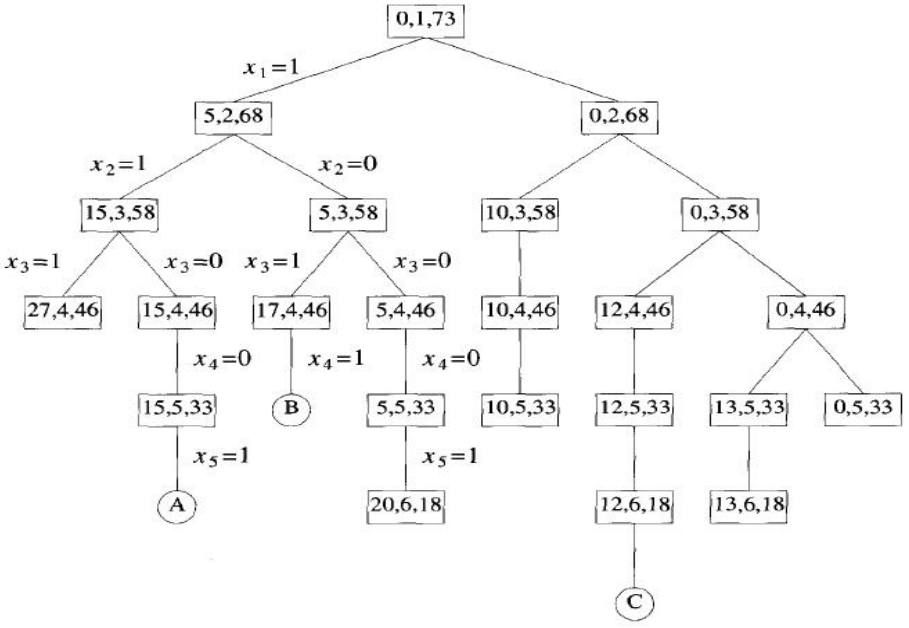


Figure 7.10 Portion of state space tree generated by SumOfSub

Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the nodes 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-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

Algorithm mcoloring (k)

```
// This algorithm 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 areassigned
// distinct integers are printed. k is the index of the next vertex to color.
{
        repeat
        {                                               // Generate all legal assignments for x[k].
                NextValue   (k);          // Assign to x [k] a legal color.
                If (x [k] = 0)  then return;       // No new color possible
                If (k =  n)  then          // at most m colors have been
                                                        // used to color the n vertices.
                        write (x [1: n]);
                        else mcoloring (k+1);
                } until (false);
        }
```

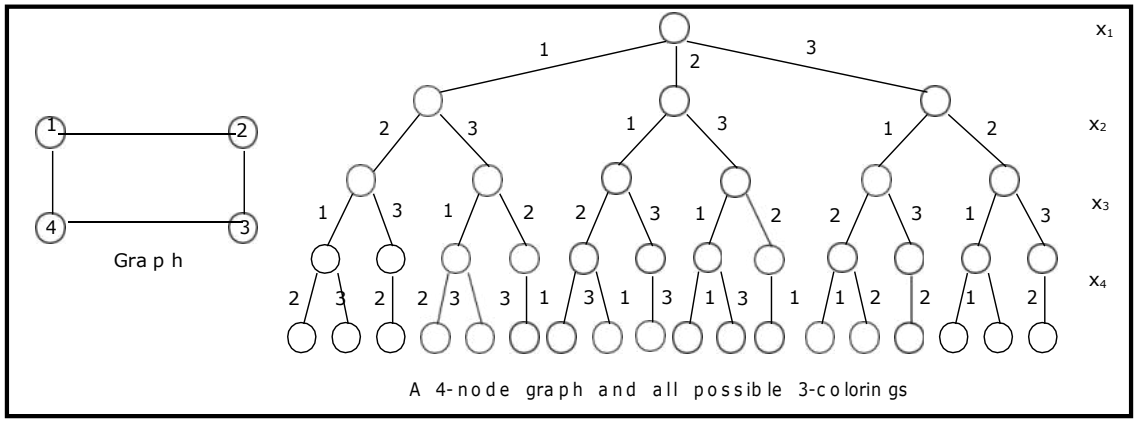
Algorithm NextValue (k)

```
// x [1] , ..... x [k-1] have been assigned integer values in the range [1, m] such that
// adjacent vertices have 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 vertices of vertex k. If no such color exists, then x [k] is 0.
{
        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
                {       // check if this color is distinct from adjacent colors
                        if ((G [k, j] ≠ 0) and (x [k] = x [j]))
                        // If (k, j) is and edge and if adj. vertices have the same color.
                        then break;
```

```
            }
            if (j = n+1)  then return;              // New color found
        } until  (false);                          // Otherwise try to find another color.
}
```

Example:

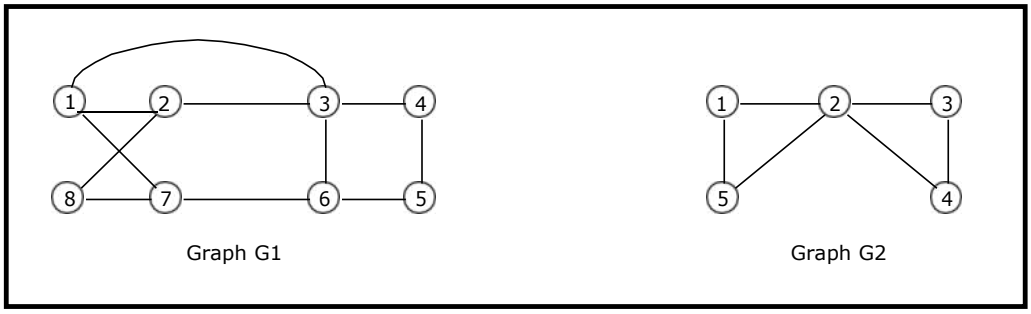
Color the graph given below with minimum number of colors by backtracking using state space tree



A 4-node graph and all possible 3-colorings

Hamiltonian Cycles:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order v_1, v_2, ... , v_{n+1}, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector (x_1, ... x_n) is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If k = 1, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from x_1, x_2, . . . , x_{k-1} and v is connected by an edge to k_{x-1}. The

UNIT V:

Branch and Bound: General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

NP-Hard and NP-Complete problems: Basic concepts, non deterministic algorithms, NP - Hard and NP Complete classes, Cook's theorem.

Branch and Bound

General method:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first

search or based on bounding function.

2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node 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 list (or queue).
- 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 (or stack).

Definition 1: Live node is a node that has been generated but whose children have not yet been generated.

Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Definition 3: Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Definition 4: Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Definition 5: The adjective "heuristic", means" related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of thetime.

Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where, $\hat{c}(x)$ is the cost of x.

h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

$\hat{g}(x)$ is an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(.)$ is called a LC–search (Least Cost search)

BFS and D-search are special cases of LC-search. If $\hat{g}(x) = 0$ and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS. If f(h(x)) = 0 and $\hat{g}(x) > \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute a estimate $\hat{c}(x)$ of c(x).

Control Abstraction for LC-Search:

Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the subtree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

A heuristic $\hat{c}(.)$ is used to estimate c(). This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$.

LC-search uses $\hat{c}$ to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

Least() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

```
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);
}
```

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E's children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

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. The three search methods differ only in the selection rule used to obtain the next E-node.

A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

A cost function $\hat{c}(.)$ 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 $c(x) \geq \hat{c}(x) >$ upper. The starting value for upper can be obtained by some heuristic or can be set to ∞.

As long as the initial value for upper is not 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.

Branch-and-bound algorithms are used for optimization problems where, 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.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function c(.), 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(.).

- 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 generally hard to compute, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x.

FIFO Branch and Bound:

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper = ∞ as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then u(2) = 19, u(3) = 14, u(4) = 18, and u(5) = 21.

The variable upper is updated to 14 when node 3 is generated. Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated. Then u(6) = 9 and so upper is updated to 9. The cost $\hat{c}(7) = 10 >$ upper and node 7 gets killed. Node 8 is infeasible and so it is killed.

Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then u(9) = 8 and so upper becomes 8. The cost $\hat{c}(10) = 11 >$ upper, and this nodeis killed.

The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $c(x)$ > upper each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(x)$ > upper are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(x)$ > upper can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and u(.) is called FIFOBB.

LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with upper $= \infty$ and node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.

As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $c(4)$ > upper and $c(5)$ > upper.

Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10$ > upper. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6.

Node 6 is the next E-node as $\hat{c}(6) = 0 < \hat{c}(3)$. Both its children are infeasible.

Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as u(9) = 8. So, node 10 with $c(10) = 11$ is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answernode.

The path = $1 \xrightarrow{2} 3 \xrightarrow{3} 9 = 5 + 3 = 8$

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph. Let C(i, J) be the cost of edge <i, j>. $c_{ij} = \infty$ if <i, j> $\notin$E and let |V| = n, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by S = {1, π, 1 | π is a

permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that (1, i_1, i_2, ... i_{n-1}, 1) $\in$ S iff <i_j, i_{j+1}> $\in$ E, 0 $\leq$ j $\leq$ n - 1 and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

   a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.

   b) Find the sum of elements, which were subtracted from rows.

   c) Apply column reductions for the matrix obtained after row reduction.

      Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

   d) Find the sum of elements, which were subtracted from columns.

   e) Obtain the cumulative sum of row wise reduction and column wise reduction.

      Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

      Associate the cumulative reduced sum to the starting state as lower bound and $\propto$ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

   a) Change all entries in row i and column j of A to $\propto$.

   b) Set A (j, 1) to $\propto$.

   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only $\propto$. Let r is the total amount subtracted to reduce the matrix.

   c) Find $c(S) = c(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $c(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $c(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

The cost matrix is $\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix $\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.

The resulting column wise reduced cost matrix (A) = $\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$

Column wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

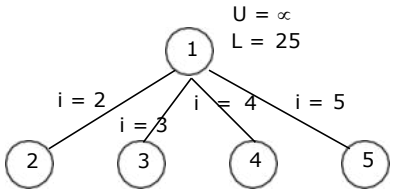
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

This is the cost of a root i.e., node 1, because this is the initially reduced costmatrix.

The lower bound for node is 25 and upper bound is $\propto$.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1,5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,2) + r$

$\hat{c}(S)$ = 25 + 10 + 0 = 35

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1\infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 3) + r$

$\hat{c}(S)$ = 25 + 17 + 11 = 53

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∞ and also set A(4, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,4) + r$

$\hat{c}(S) = 25 + 0 + 0 = 25$

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∞ and also set A(5, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

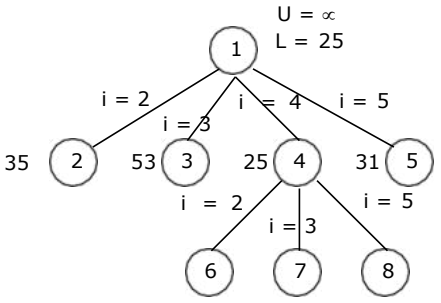
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,5) + r$

$\hat{c}(S) = 25 + 1 + 5 = 31$

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∞ and also set A(2, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(4, 2) + r$

$c(S)$ = 25 + 3 + 0 = 28

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∞ and also set A(3, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $c(S) = c(R) + A(4, 3) + r$

$c(S)$ = 25 + 12 + 13 = 50

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to ∞ and also set A(5, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

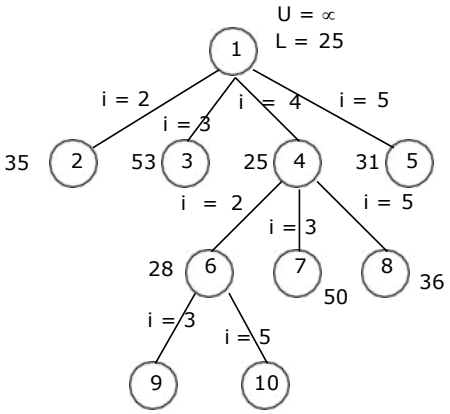
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 11
Column reduction sum = 0
Cumulative reduction (r) = 11+0 = 11

Therefore, as $c(S) = c(R) + A(4, 5) + r$

$c(S)$ = 25 + 0 + 11 = 36

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A(3, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2, 3) + r$

$\hat{c}(S)$ = 28 + 11 + 13 = 52

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5, 1) to ∞.

$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2, 5) + r$

$\hat{c}(S)$ = 28 + 0 + 0 = 28

The tree organization up to this point is as follows:

UNIT-5

NP Hard and NP Complete Problems: Basic Concepts, Cook's theorem.

NP Hand Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP)

NP Hand Scheduling Problems. Scheduling Identical Processors, Job Shop Scheduling

NP-Hard and NP-Complete Problems: Basic Concepts

1. P vs NP: The Starting Point

- Class P: Problems that can be solved by a deterministic algorithm in polynomial time. In simpler terms, these are problems where the solution can be found efficiently.

- Example: Sorting numbers using bubble sort, finding the shortest path in a graph (Dijkstra's algorithm).

- Class NP: Stands for nondeterministic polynomial time. These are problems where the solution might be hard to find, but easy to verify in polynomial time.

- Example: Sudoku—given a solution, it's easy to verify its correctness, but finding that solution might take a lot of time.

2. NP-Hard Problems

- Definition: A problem is NP-Hard if every problem in NP can be reduced to it in polynomial time.

- Key Point: NP-Hard problems don't have to be in NP, meaning their solution might not be verifiable in polynomial time. They are at least as hard as the hardest NP problems.

- Solving an NP-Hard problem would mean you could solve all NP problems.

- Examples of NP-Hard Problems:

- Travelling Salesman Problem (TSP): Given a list of cities and the distances between each pair, find the shortest possible route that visits each city once and returns to the origin city.

- Knapsack Problem (decision version): Given a set of items, each with a weight and value, determine if there is a subset of the items whose total weight does not exceed a given limit and whose total value is at least a specified amount.

3. NP-Complete Problems

- Definition: A problem is NP-Complete if:
  1. It is in NP (the solution can be verified in polynomial time).
  2. It is NP-Hard, meaning any NP problem can be reduced to it in polynomial time.

- Key Point: If an NP-Complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time, i.e., P = NP.

- Examples of NP-Complete Problems:
  - Subset Sum Problem: Given a set of integers, is there a non-empty subset whose sum is zero?
  - 3-SAT Problem: A logical formula in conjunctive normal form with three literals per clause. Determine if there exists an assignment of variables such that the formula evaluates to true.

4. Cook's Theorem

- Cook's Theorem (1971) is a foundational result in computational complexity theory. It was the first to show that a problem, specifically Boolean Satisfiability Problem (SAT), is NP-Complete.

- Key Points of the Theorem:
  - It shows that SAT (the problem of determining whether a given Boolean formula can be satisfied by some assignment of true/false values to its variables) is NP-Complete.
  - This was the first proof that there are problems for which solving one would mean you could solve all NP problems.

- Implication: Cook's Theorem established the concept of NP-Complete problems and laid the foundation for showing that other problems (like 3-SAT, Knapsack, etc.) are NP-Complete. Essentially, it provided a universal method for reducing any NP problem to another NP-Complete problem in polynomial time.

5. Reductions and Problem Transformation

- To prove a problem is NP-Complete, we use the concept of polynomial-time reduction. This means we transform one NP problem into another in polynomial time. If we can reduce a known

NP-Complete problem (like 3-SAT) to another problem in NP, then the new problem is NP-Complete.

6. Summary of Relationships

- NP-Hard: Problems as hard as NP problems but not necessarily in NP (no efficient solution or even verification).
- NP-Complete: Problems in NP (verifiable in polynomial time) and as hard as the hardest NP problems (any NP problem can be transformed into an NP-Complete problem).

- If you solve an NP-Complete problem in polynomial time, then you solve all NP problems in polynomial time, and it would imply that P = NP—a major unsolved question in computer science.

Visual Summary

- P ⊆ NP
- NP-Complete ⊆ NP ⊆ NP-Hard
- NP-Hard: Not necessarily in NP but as hard as NP problems.
- NP-Complete: Problems that are both in NP and NP-Hard.

Cook’s Theorem:

Cook's Theorem is one of the most important results in computer science, and it establishes the concept of NP-Completeness. In simple terms, it tells us that Boolean Satisfiability Problem (SAT) is the first known NP-Complete problem.

Let’s break it down step by step.

1. What is SAT?

The Boolean Satisfiability Problem (SAT) asks whether a logical expression (a collection of AND, OR, and NOT operations) can be made true by assigning values (true/false) to its variables.

For example:

- (x1 OR NOT x2) AND (x2 OR x3)

The question is: can you assign true/false to the variables (x1), (x2), and (x3) so that the whole expression evaluates to true?

In this case:

- If we set (x1 = {true}), (x2 = {false}), and (x3 = {true}), the expression becomes true.

2. What does NP mean?

A problem is in NP if, given a solution, you can verify whether it is correct in polynomial time (efficient time).

For SAT:

- If someone gives you a set of variable assignments (like x1 = {true} , x2 = {false}), and (x3 = {true})), you can easily check whether the expression becomes true or not in polynomial time. So, SAT is in NP.

3. What is NP-Complete?

- A problem is NP-Complete if:

1. It is in NP.
2. Every other NP problem can be transformed (or reduced) into it in polynomial time.

This means that solving one NP-Complete problem efficiently (in polynomial time) would allow you to solve all NP problems efficiently.

4. What Does Cook's Theorem Say?

Cook's Theorem says that SAT is NP-Complete. It means:

1. SAT is in NP (as we discussed).
2. Every other NP problem can be reduced to SAT in polynomial time.

5. Why Is This Important?

This was the first time someone (Stephen Cook) showed that there exists a problem (SAT) that is as hard as any problem in NP. If we can solve SAT efficiently, we can solve every NP problem efficiently, which means P = NP (a huge question in computer science).

6. Example to Understand Cook's Theorem

Let's simplify it with an analogy:

Imagine you have a magic box that can solve SAT problems instantly. Now, Cook's Theorem tells us that for any other NP problem (like Sudoku, Knapsack, etc.), we can transform that problem into a SAT problem and use this magic box to solve it.

For example:

- Knapsack Problem: You need to find the best combination of items to put in a knapsack without exceeding the weight limit.
- Using Cook's Theorem, you can take the Knapsack Problem, convert it into a SAT problem (a Boolean expression), and then use the SAT solver (magic box) to find a solution.

This means if we had an efficient algorithm to solve SAT, we could solve any NP problem using it.

NP Hard Graph Problems:

Clique Decision Problem (CDP)

The Clique Decision Problem (CDP) is one of the well-known NP-Hard graph problems

1. What is a Clique?

In a graph, a clique is a subset of vertices where every pair of vertices is connected by an edge. In simpler terms, it's a group of nodes where each node is directly connected to every other node in the group.

2. What is the Clique Decision Problem (CDP)?

The Clique Decision Problem asks:

- Given a graph and a number (k), is there a clique of size (k) or larger in the graph?

This means you need to find whether there's a group of (k) nodes that are all connected to each other. If such a group exists, the answer is "yes," otherwise, it's "no."

3. Why is CDP NP-Hard?

- Verification: If someone gives you a subset of vertices and says it's a clique, you can easily check if every pair of vertices in the subset is connected (this takes polynomial time). So, it is in NP.
- Hardness: Finding this clique is difficult because, in the worst case, you might have to check many combinations of nodes to see if they form a clique.

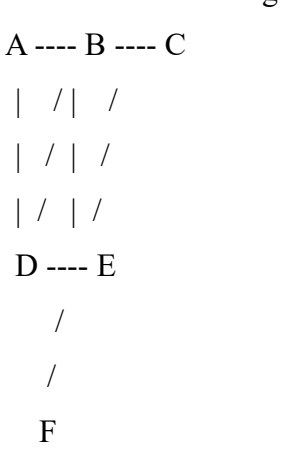
Since finding a clique of size (k) can't be done efficiently for large graphs (we don't have a known polynomial-time solution), it is classified as an NP-Hard problem. If we could solve CDP efficiently, we could solve all NP problems efficiently.

4. Example of the Clique Decision Problem (CDP)

Let's walk through a simple example to understand the problem.

Example Graph:

Consider the following graph with 6 vertices (labeled (A, B, C, D, E, F)):



Problem Statement:

We are asked: Is there a clique of size 3 or larger in this graph?

1. Step 1: Check Possible Subsets of Vertices

We need to check if there are any groups of 3 or more vertices where every vertex is connected to every other vertex in the group.

2. Step 2: Examine Different Groups

- Let's check the subset ({A, B, D}):
  - Are all nodes connected?
    - A is connected to B.
    - A is connected to D.
    - B is connected to D.
  - Yes, all pairs are connected! So, ({A, B, D}) is a clique of size 3.
  - You can also check other groups like ({B, C, E}) and find out if they form cliques (in this case, they do not).

3. Step 3: Answer

Since we found a clique of size 3 (the set ({A, B, D})), the answer to the Clique Decision Problem for (k = 3) is yes.

5. Key Points About the Clique Decision Problem (CDP)

- Input: A graph (G) and a number (k).
- Output: “Yes” if there exists a clique of size (k) or more in the graph, otherwise “No.”
- NP-Hardness: Even though it’s easy to verify a solution if someone gives us the subset, finding that clique of size (k) is computationally hard, especially as the size of the graph grows. That’s why CDP is NP-Hard.

6. Why Is This Problem Important?

The Clique Decision Problem is important in areas like:

- Social Networks: Finding tightly connected groups of people (cliques).
- Bioinformatics: Identifying closely related genes or proteins.
- Network Analysis: Identifying dense sub-networks.

Since it’s NP-Hard, no efficient algorithm is known to solve CDP for large graphs, which means finding cliques is challenging for large datasets.

Chromatic Number Decision Problem (CNDP)

The Chromatic Number Decision Problem (CNDP) is a classic NP-Hard problem in graph theory. Let’s break it down in very simple terms and use a clear example to understand it.

1. What is Graph Coloring?

Graph coloring is the process of assigning colors to the vertices of a graph such that:

- No two adjacent vertices (vertices connected by an edge) have the same color.

2. What is the Chromatic Number?

The chromatic number of a graph is the minimum number of colors needed to color the graph properly (with no two adjacent vertices sharing the same color).

3. What is the Chromatic Number Decision Problem (CNDP)?

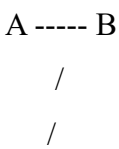
The Chromatic Number Decision Problem asks a simple question:

- Given a graph and a number (k), can you color the graph with (k) or fewer colors?

In other words, is it possible to color the graph using (k) colors so that no two connected vertices have the same color?

4. Example of CNDP

Consider the following graph with 5 vertices:



C ----- D

E

Problem Statement:

Let's say we are asked: Can you color this graph using 3 colors or fewer?

1. Step 1: Try to Color the Graph with 3 Colors

We need to assign colors to the vertices such that:

- No two connected vertices (like A and B) get the same color.

2. Step 2: Color the Graph

- Let's assign Color 1 to vertex A.
- Vertex B is adjacent to A, so it can't have Color 1. We give it Color 2.
- Vertex C is adjacent to both A and B, so it needs a different color. We give it Color 3.
- Vertex D is adjacent to B and C, so we can give it Color 1 (since it's not adjacent to A, which already has Color 1).
- Vertex E is adjacent to C, so it can't have Color 3. We can assign it Color 2 (since it's not adjacent to B, which has Color 2).

3. Step 3: Check if the Graph is Properly Colored

After coloring the graph, here's what we have:

- A = Color 1
- B = Color 2
- C = Color 3
- D = Color 1
- E = Color 2

No two connected vertices have the same color, so this is a valid coloring with 3 colors.

4. Step 4: Answer

Since we successfully colored the graph with 3 colors, the answer to the CNDP for (k = 3) is yes.

5. Why is CNDP NP-Hard?

The Chromatic Number Decision Problem is NP-Hard because:

- Verification: If someone gives you a colored graph, it's easy to check if the coloring is correct (just check if adjacent vertices have different colors). This can be done in polynomial time.

- Hardness: However, finding the chromatic number (the minimum number of colors) is difficult because you have to try different combinations of color assignments, especially for large graphs.

Since we don't know of any efficient way to solve CNDP for large graphs, it is classified as NP-Hard.

6. Key Points About the Chromatic Number Decision Problem (CNDP)

- Input: A graph and a number (k).
- Output: "Yes" if the graph can be colored with (k) colors or fewer, and "No" if it cannot be.
- NP-Hardness: While verifying a coloring is easy, finding the chromatic number is computationally hard, which is why CNDP is NP-Hard.

7. Why Is This Problem Important?

Graph coloring problems like CNDP are important in areas like:

- Scheduling: Assigning time slots to tasks or exams without conflicts (e.g., no two tasks sharing the same resource at the same time).
- Map Coloring: Ensuring that no two neighboring regions on a map share the same color.
- Resource Allocation: Assigning resources such that no two conflicting tasks use the same resource.

Since it's NP-Hard, CNDP is difficult to solve efficiently for large graphs, which means finding the chromatic number is a computational challenge in many practical applications.

Traveling Salesperson Decision Problem (TSP) –

The Traveling Salesperson Problem (TSP) is a famous NP-Hard problem in computer science. It asks a very straightforward question about finding the shortest possible route for a salesperson who needs to visit several cities and return to the starting point.

1. What is the TSP Problem?

In the Traveling Salesperson Problem (TSP), you're given a list of cities and the distances between every pair of cities. The goal is to find the shortest route that:

- Starts at one city,
- Visits each city exactly once,
- And returns to the starting city.

2. What is the Traveling Salesperson Decision Problem (TSP Decision Problem)?

The TSP Decision Problem asks a yes-or-no question:

- Given a set of cities, a set of distances between them, and a number (k), is there a route that visits all the cities exactly once and has a total distance of (k) or less?

In simple terms, it asks whether you can find a tour of the cities where the total distance does not exceed a specified number (k).

3. Why is the TSP Decision Problem NP-Hard?

The TSP Decision Problem is NP-Hard because:

- Verification: If someone gives you a solution (a route), it's easy to check whether the total distance of that route is less than or equal to (k). You just add up the distances. This can be done in polynomial time.

- Hardness: Finding the actual shortest route (or even determining if a route exists within distance (k)) is very difficult. For large sets of cities, the number of possible routes grows exponentially, and there's no known efficient algorithm to solve the problem in polynomial time.

Because finding the solution is computationally hard, TSP is classified as NP-Hard.

4. Example of TSP Decision Problem

Let's go through a simple example to understand how the TSP Decision Problem works.

Example:

Imagine a salesperson needs to visit 4 cities: A, B, C, and D, and the distances between them are as follows:

- Distance from A to B: 10 km
- Distance from A to C: 15 km
- Distance from A to D: 20 km
- Distance from B to C: 35 km
- Distance from B to D: 25 km
- Distance from C to D: 30 km

Problem Statement:

The question is: Is there a route that visits all the cities (A, B, C, and D) exactly once and returns to the starting point, with a total distance of 80 km or less?

1. Step 1: Consider Different Routes

To solve this problem, you would have to check different possible routes and see if any of them have a total distance of 80 km or less.

Some possible routes are:

- Route 1: A → B → C → D → A
- Route 2: A → B → D → C → A
- Route 3: A → C → B → D → A

2. Step 2: Calculate the Total Distance

- For Route 1 (A → B → C → D → A):
  - A → B = 10 km
  - B → C = 35 km
  - C → D = 30 km
  - D → A = 20 km
  - Total = 10 + 35 + 30 + 20 = 95 km
- For Route 2 (A → B → D → C → A):
  - A → B = 10 km
  - B → D = 25 km
  - D → C = 30 km
  - C → A = 15 km
  - Total = 10 + 25 + 30 + 15 = 80 km

In this case, Route 2 has a total distance of exactly 80 km.

3. Step 3: Answer

Since we found a route (Route 2) that visits all the cities and returns to the starting point with a total distance of 80 km, the answer to the TSP Decision Problem for (k = 80) is yes.

5. Key Points About the Traveling Salesperson Decision Problem (TSP Decision Problem)

- Input: A set of cities, the distances between them, and a number (k).
- Output: "Yes" if there's a route that visits all cities exactly once and returns to the starting point with a total distance of (k) or less, otherwise "No."
- NP-Hardness: While it's easy to verify a solution (checking if a route's total distance is within the limit), finding the route itself is hard because the number of possible routes grows exponentially with the number of cities.

6. Why Is TSP Important?

The Traveling Salesperson Problem is important in many real-world applications, including:

- Logistics and Delivery: Finding the most efficient routes for delivering goods.
- Manufacturing: Optimizing the order of operations for machines in a factory.
- Computer Networks: Efficiently routing data between nodes in a network.

Since it's NP-Hard, finding efficient solutions for large-scale TSP instances is challenging, and this is why it's studied extensively in optimization and computer science.

NP Hard Scheduling Problems:

Scheduling Identical Processors

The Scheduling Problem for Identical Processors is a classic example of an NP-Hard problem in the area of task scheduling. Let's break it down step by step in a simple and understandable way with an example.

1. What is the Scheduling Problem for Identical Processors?

In this problem, we have:

A set of tasks (or jobs), each with a specific processing time.

A set of identical processors (machines) that can run the tasks.

The goal is to assign the tasks to the processors such that:

The total processing time (or load) is balanced as evenly as possible across all processors.

We want to minimize the makespan, which is the time it takes for the longest-running processor to finish all its assigned tasks.

2. Why is This Scheduling Problem NP-Hard?

Verification: If someone gives you a schedule, it's easy to check how long each processor takes to finish and verify the total processing time.

Hardness: Finding the best (optimal) schedule where the makespan is minimized is hard because there are many possible ways to assign the tasks, especially when there are many tasks and processors.

There's no known efficient (polynomial-time) algorithm to find the optimal solution, which makes this problem NP-Hard.

3. Example: Scheduling Tasks on Identical Processors

Let's go through a simple example with 4 tasks and 2 processors to make this clear.

Tasks and Their Processing Times:

Task 1: 5 units of time

Task 2: 2 units of time

Task 3: 3 units of time

Task 4: 7 units of time

Problem Statement:

You have 2 identical processors and need to schedule these tasks on them. How should you assign the tasks to the processors to minimize the makespan?

Step 1: List the Tasks and Processors

You have 2 processors: Processor 1 and Processor 2.
You have 4 tasks: Task 1 (5 units), Task 2 (2 units), Task 3 (3 units), and Task 4 (7 units).
Step 2: Assign Tasks to Processors Let's try assigning tasks to processors in different ways and calculate the makespan for each:

Option 1: Assign Task 1 and Task 4 to Processor 1, and Task 2 and Task 3 to Processor 2.

Processor 1: Task 1 (5 units) + Task 4 (7 units) = 12 units of time
Processor 2: Task 2 (2 units) + Task 3 (3 units) = 5 units of time
Makespan: The longest processing time is 12 units (Processor 1).
Option 2: Assign Task 1 and Task 2 to Processor 1, and Task 3 and Task 4 to Processor 2.

Processor 1: Task 1 (5 units) + Task 2 (2 units) = 7 units of time
Processor 2: Task 3 (3 units) + Task 4 (7 units) = 10 units of time
Makespan: The longest processing time is 10 units (Processor 2).
Option 3: Assign Task 2 and Task 4 to Processor 1, and Task 1 and Task 3 to Processor 2.

Processor 1: Task 2 (2 units) + Task 4 (7 units) = 9 units of time
Processor 2: Task 1 (5 units) + Task 3 (3 units) = 8 units of time
Makespan: The longest processing time is 9 units (Processor 1).
Step 3: Find the Optimal Schedule Among the three options we tried:

Option 3 has the smallest makespan (9 units of time).
So, the best way to assign tasks to the processors in this case is:

Processor 1: Task 2 (2 units) and Task 4 (7 units)
Processor 2: Task 1 (5 units) and Task 3 (3 units)
Step 4: Answer The minimum makespan for this task set on two processors is 9 units of time.

4. Why Is This Problem NP-Hard?

The reason this scheduling problem is NP-Hard is that there are many ways to assign the tasks to the processors. As the number of tasks increases, the number of possible assignments grows exponentially, making it very difficult to find the optimal solution in a reasonable amount of time for large instances.

Even for small examples, we had to try different assignments and check the makespan for each one. In larger problems with more tasks and processors, this process becomes infeasible without a good heuristic or approximation algorithm.

5. Key Points About Scheduling Identical Processors

Goal: Assign tasks to identical processors so that the processing time is balanced, and the makespan (longest processing time) is minimized.

NP-Hardness: Finding the optimal schedule is computationally difficult, even though verifying a given schedule is easy.

Makespan: The time it takes for the slowest processor to finish all its tasks.

6. Why Is This Problem Important?

The Scheduling Problem for Identical Processors is important in many real-world scenarios, such as:

Manufacturing: Scheduling jobs on machines to minimize production time.

Computer Systems: Allocating tasks to processors in a multi-core system to optimize performance.

Logistics: Distributing workloads to workers or vehicles in a balanced way to minimize total operation time.

Because it's NP-Hard, there are no easy solutions for large instances, and this is why approximation algorithms and heuristics are often used in practice to solve it.

Job Shop Scheduling

Job Shop Scheduling (JSS) is a classic NP-Hard problem in operations research and computer science. It involves scheduling jobs with multiple tasks on different machines in such a way that the total time to complete all jobs (called the makespan) is minimized.

1. What is Job Shop Scheduling?

In the Job Shop Scheduling Problem, you are given:

- Jobs: Each job is made up of a sequence of tasks.
- Machines: Each task must be processed on a specific machine, and different jobs may need the same machine.
- Goal: The aim is to schedule the tasks so that the jobs are completed as quickly as possible while minimizing the makespan (the total time to finish all jobs).

The challenge is to decide the order in which to process the tasks on each machine, ensuring no machine works on more than one task at a time, and each job follows its required sequence of tasks.

2. Why is Job Shop Scheduling NP-Hard?

Job Shop Scheduling is NP-Hard because:

- Verification: Given a schedule, it's easy to check the total makespan (how long all jobs take to finish) and verify if the schedule is valid.
- Hardness: Finding the optimal schedule is very difficult. As the number of jobs and machines increases, the number of possible schedules grows exponentially, making it computationally expensive to find the best solution.

Because of this complexity, Job Shop Scheduling is classified as an NP-Hard problem.

3. Example of Job Shop Scheduling

Let's go through a simple example to better understand how Job Shop Scheduling works.

Problem Setup:

- 2 Jobs: Job 1 and Job 2.
- 3 Machines: Machine A, Machine B, and Machine C.
- Each job has tasks that must be done in a specific order on certain machines.

Task Details:

Job 1:

- Task 1.1: Process on Machine A for 2 units of time.
- Task 1.2: Process on Machine B for 3 units of time.
- Task 1.3: Process on Machine C for 2 units of time.

Job 2:

- Task 2.1: Process on Machine B for 4 units of time.
- Task 2.2: Process on Machine A for 1 unit of time.
- Task 2.3: Process on Machine C for 3 units of time.

Problem Statement:

We need to schedule the tasks on the machines in such a way that all tasks are completed as soon as possible, minimizing the makespan.

4. Steps to Solve the Example

Step 1: Visualize the Tasks

We can visualize the task dependencies as follows:

Job 1:

Task 1.1 must be completed before Task 1.2, and Task 1.2 must be completed before Task 1.3.

Job 2:

Task 2.1 must be completed before Task 2.2, and Task 2.2 must be completed before Task 2.3.

Step 2: Assign Tasks to Machines

Let's start assigning tasks to the machines, keeping in mind that no two tasks can be processed on the same machine at the same time.

Machine A:

First, we can process Task 1.1 (Job 1) for 2 units of time.

After that, we can process Task 2.2 (Job 2) for 1 unit of time.

Machine B:

We can process Task 2.1 (Job 2) for 4 units of time first.

Then, process Task 1.2 (Job 1) for 3 units of time.

Machine C:

Once Task 1.2 is finished, we can process Task 1.3 (Job 1) for 2 units of time.

After Task 2.2 is completed, we can process Task 2.3 (Job 2) for 3 units of time.

Step 3: Calculate the Makespan

Let's look at the schedule on each machine:

Machine A:

Task 1.1 from time 0 to 2.

Task 2.2 from time 2 to 3.

Machine B:

Task 2.1 from time 0 to 4.

Task 1.2 from time 4 to 7.

Machine C:

Task 1.3 from time 7 to 9.

Task 2.3 from time 3 to 6.

The makespan is determined by the longest time any machine is active. In this case, Machine C finishes last at time 9 units, so the total makespan is 9 units of time.

5. Challenges and NP-Hardness

In this example, we manually assigned the tasks to machines and calculated the makespan, but for larger problems with more jobs and machines, the number of possible schedules becomes huge. Finding the best (optimal) schedule is computationally expensive, which is why Job Shop Scheduling is NP-Hard.

There's no known efficient (polynomial-time) algorithm to solve the problem optimally for large instances, which is why it's classified as NP-Hard.

6. Key Points about Job Shop Scheduling

- Multiple Jobs and Tasks: Each job consists of several tasks that must be processed on specific machines in a specific order.
- Machines: The same machine can't process more than one task at a time.
- Goal: Minimize the makespan, or the total time to complete all jobs.
- NP-Hardness: It's easy to verify a solution, but finding the optimal schedule is very difficult due to the exponential number of possibilities as the number of jobs and machines grows.

7. Why Is Job Shop Scheduling Important?

The Job Shop Scheduling Problem is important in real-world applications such as:

- Manufacturing: Where multiple jobs need to be processed on different machines, and scheduling them efficiently reduces production time and costs.
- Project Management: Where tasks need to be allocated to workers or resources in an efficient manner.
- Computer Systems: Scheduling processes or tasks in distributed computing environments.