SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES

(AUTONOMOUS)

Dr. Visweswaraiah Road, (Bangalore-Tirupathi Bye-pass Road),
Murukambattu,Chittoor – 517127, Andhra Pradesh, India.

Subject Name: ADVANCED DATA STRUCTURES & ALGORITHM ANALYSIS

Year & Semester: II B.Tech. - III Semester

Faculty Name: Mrs.S.Anusha, Assistant Professor

Department of Computer Science and Engineering (Artificial Intelligence)

II B.Tech. - III Semester

23CSE231 ADVANCED DATA STRUCTURES & ALGORITHM ANALYSIS L T P C
3 0 0 3

PRE-REQUISITES: A course on C and Data Structures

COURSE EDUCATIONAL OBJECTIVES:

1. To provide knowledge on advance data structures frequently used in Computer Science domain
2. To develop skills in algorithm design techniques popularly used
3. To understand the use of various data structures in the algorithm design

UNIT I: INTRODUCTION (9)

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.

UNIT 2: HEAP TREES & DIVIDE AND CONQUER (9)

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.

UNIT 3: GREEDY METHOD & DYNAMIC PROGRAMMING (9)

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.

UNIT 4: BACKTRACKING & BRANCH AND BOUND (9)

Backtracking: General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, 0/1 Knapsack Problem.

Branch and Bound: The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem.

UNIT 5: NP HARD, NP HARD GRAPH PROBLEMS & NP HARD SCHEDULING PROBLEMS (9)

NP Hard and NP Complete Problems: Basic Concepts, Cook's theorem

NP Hard Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP)

NP Hard Scheduling Problems: Scheduling Identical Processors, Job Shop Scheduling.

Total Hours: 45

COURSE OUTCOMES:

On successful completion of the course, students will be able to		Pos
CO1	Illustrate the working of the advanced tree data structures and their applications	**PO1, PO2**
CO2	Understand the Graph data structure, traversals and apply them in various contexts.	**PO1, PO2, P03, PO4**
CO3	Use various data structures in the design of algorithms	**PO1, PO2, P03, PO4, PO5**
CO4	Recommend appropriate data structures based on the problem being solved.	**PO1, PO2, P03, PO4, PO5**
CO5	Analyze algorithms with respect to space and time complexities	**PO1, PO2, P03, PO4**
CO6	Design new algorithms	**PO1, PO2, P03, PO4, PO5**

TEXT BOOKS:

1. Fundamentals of Data Structures in C++, Horowitz, Ellis; Sahni, Sartaj; Mehta, Dinesh 2nd Edition Universities Press.
2. Computer Algorithms / C++ Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, 2nd Edition University Press.

REFERENCE BOOKS:

1. Data Structures and program design in C, Robert Kruse, Pearson Education Asia
2. An introduction to Data Structures with applications, Trembley & Sorenson, McGraw Hill
3. The Art of Computer Programming, Vol.1: Fundamental Algorithms, Donald E Knuth, Addison-Wesley, 1997.
4. Data Structures using C & C++: Langsam, Augenstein&Tanenbaum, Pearson, 1995
5. Algorithms + Data Structures & Programs:, N.Wirth, PHI
6. Fundamentals of Data Structures in C++: Horowitz Sahni& Mehta, Galgottia Pub.
7. Data structures in Java:, Thomas Standish, Pearson Education Asia

REFERENCE WEBSITE:

1. https://www.tutorialspoint.com/advanced_data_structures/index.asp
2. http://peterindia.net/Algorithms.html
3. Abdul Bari,1. Introduction to Algorithms (youtube.com)

CO-PO MAPPING:

CO\PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	2	-	-	-	-	-	-	-	-	-	-
CO2	3	3	2	2	-	-	-	-	-	-	-	-
CO3	3	2	2	2	2	-	-	-	-	-	-	-
CO4	3	2	2	2	2	-	-	-	-	-	-	-
CO5	3	2	2	2	-	-	-	-	-	-	-	-
CO6	3	2	2	2	2	-	-	-	-	-	-	-
CO*	**3**	**2**	**2**	**2**	**2**	**-**	**-**	**-**	**-**	**-**	**-**	**-**

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

1.1 Introduction to Algorithm Analysis

An algorithm is an effective method for finding out the solution for a given problem. It is a sequence of instruction. That conveys the method to address a problem.

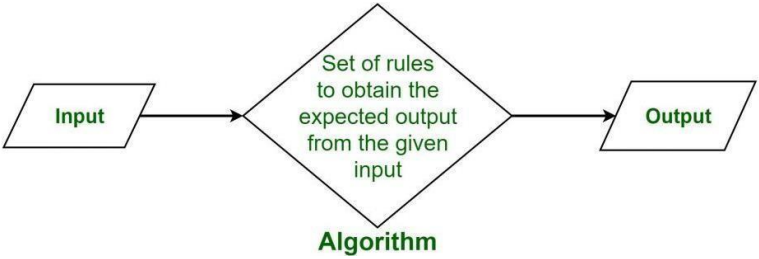
1.1.1 Definition of Algorithm

Algorithm: an algorithm is a step by step procedure to solve a computational problem is called an algorithm.

or

A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm.

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



1.1.2 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:

a) 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.

b) 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.

c) Operations Research: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.

d) 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.

e) 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.

1.1.3 What is the need for algorithms?

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

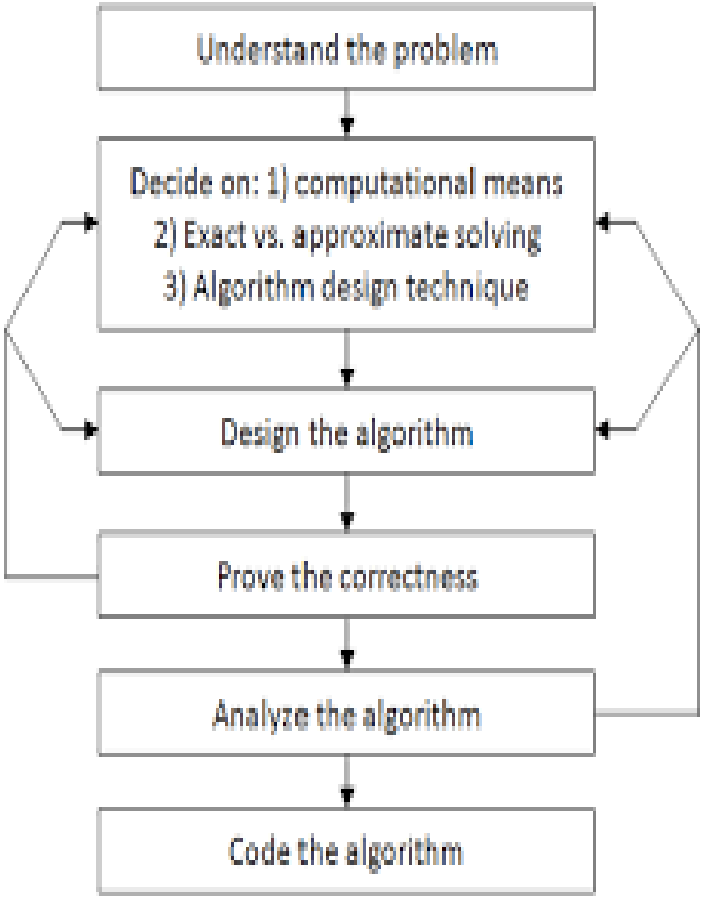
1.1.4 Properties of an Algorithm

According to D.E. Knuth a pioneer in the computer science discipline an algorithm must have the following properties

1. **INPUT:** The Algorithm should be given zero or more input.
2. **OUTPUT:** At least one quantity is produced. For each input the algorithm produced value from specific task.
3. **DEFINITENESS:** Each instruction is clear and unambiguous.
4. **FINITENESS:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **EFFECTIVENESS:** Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

6. **Correctness**
   An algorithm must produce the correct output values for all legal input instances of the problem
7. **Generality**
   The algorithm should be applicable to all problems of a similar form
8. **Multiple view**
   Same algorithm may be represented in different ways
9. **Multiple Availability**
   Several algorithms for solving the same problem may exist - with different properties

1.1.5 Process for design and analysis of algorithms



1. **Understand the problem:** The first thing you need to do before designing an algorithm is to understand the problem completely.this is a crucial phase,so we should be very careful.If we did any mistake in this step the entire algorithm becomes wrong.

2. **Exact vs approximate solving:** Solve the problem exactly if possible, otherwise use approximation methods. Even though some problems are solvable by exact method, but they are not faster when compared to approximation method. So in that situation, we will use approximation method.

3. **Algorithm Design Techniques:** In this we will use different design techniques like:
   a) Divide and conquer
   b) Greedy method
   c) Dynamic programming
   d) Backtracking
   e) Branch and bound
   f) Brute force
   g) Decrease and conquer
   h) Transform and conquer
   i) Space and Time trade-offs

Depending upon the problem, we will use a suitable design method.

4. **Prove Correctness:** Once an algorithm has been specified, next we have to prove its correctness. Usually testing is used for proving correctness.

5. **Analyze an Algorithm:** Analyzing the algorithm means studying the algorithm behaviour i.e., calculating the time complexity and space complexity. If the time complexity of algorithm is more, then we will use one more designing technique such that time complexity should be minimum.

6. **Coding an Algorithm:** After completion of all phases successfully, then we will code an algorithm. Coding should not depend on any program language.We will use general notation (pseudo code) and English language statement. Ultimately algorithms are implemented as computer programs.

Pseudo code for expressing algorithms

The general procedure of writing the pseudo code is presented below:

1. Single line comments are written using // as beginning of comment.
2. The beginning and end of block should be indicated by { and } respectively. The compound statements should be enclosed within { and } brackets.

   Example:

```
{
  stmt1;
  stmt2;
 }
```

3. All statements are separated by ; (semicolon).

4. The identifier should begin by a letter and not by digit. An identifier can be a combination of alphanumeric string.

Example:
var1, var2, x, y;

5. Assignment of values to the variables is done using the assignment operator :=

   Example:
   x := 2

6. There are two boolean values: TRUE and FALSE.
   Logical operators – AND, OR, NOT
   Relational operators – <, <=, =, ≠, >, >=

7. The conditional statement if-then & if-then-else are written in the following form:

   if (condition) then statement
   if (condition) then statement1 else statement2

Here condition is a boolean expression and statement1, statement2 are either simple or compound statements.

8. Case Statement

```
case
{
  (condition 1): statement1;
  (condition 2): statement2;
   ...
  (condition n): statementn;
   ...
   else: (statementn+1);
}
```

If condition1 is true, statement1 is executed and the case statement is exited.
If statement1 is false, condition2 is evaluated.

If condition2 is true, statement2 is executed and so on.
If none of the conditions are true, statement(n+1) is executed and loop is exited.
The else clause is optional.

9. Looping Statements

a) For Loop

The general form for writing for loop is:

```
for variable := value1 to value n step do
{
   stmt1;
   stmt2;
   ...
   stmtn;
}
```

Here value1 is initialization condition and valuen is a terminating condition.
The step indicates the increments or decrements in value for executing the for loop.

b) While Loop

While statement can be written as:

```
while (condition) do
{
   stmt1;
   stmt2;
   ...
   stmtn;
}
```

As long as condition is true, the statement gets executed otherwise the loop is exited. The value of condition is evaluated at the beginning of the loop.

10. The break & return can be used within any of the above

instructions to exit from the loop.

11. Algorithm is a procedure consisting of heading and body.
The heading has the form:

Algorithm Name(parameter list)

Where Name is the name of the procedure and parameter list is a listing of the procedure parameters.
The body has one or more simple or compound statements enclosed within braces { and }.

Example 1: Write an algorithm to count the sum of n numbers

```
Algorithm sum(i, n)
{
   sum := 0;
   for i := 1 to n do
      i := i + 1
      sum := sum + i;
}
```

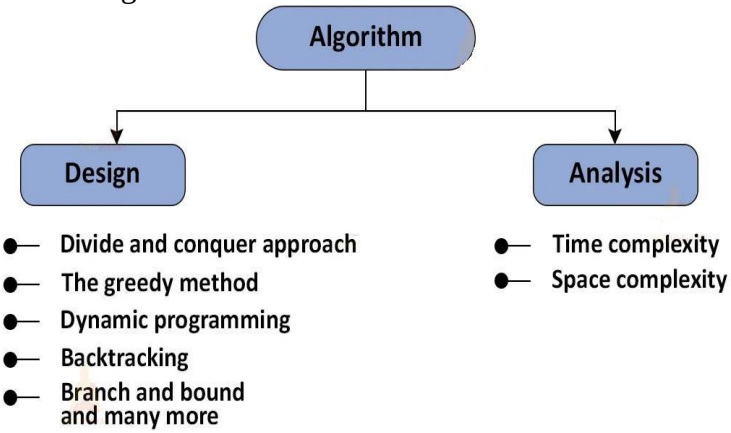
Example 2: Write an algorithm to check whether a given number is even or odd

```
Algorithm evenOdd(value)
{
   if (value % 2 = 0) then
      write "number is even";
   else
      write "number is odd";
}
```

11.1 Overview of time and space complexity analysis

Performance Analysis: The efficiency of an algorithm can be decided by measuring the performance of algorithm. We can measure the performance of an algorithm by computing amount of time and storage requirement. We can analyse an algorithm by two ways.

1. By checking the correctness of an algorithm.
2. By measuring time and space complexity of an algorithm.



Time Complexity

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input.

Example

Statements	**s/e**	**Frequency**	**Total steps**
Algorithm sum(a,n)	0	-	0
{	0	-	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i]	1	n	n
return s;	1	1	1
}	0	-	0
			2n+3

Space Complexity

The amount of memory used by a program to execute it is represented by its space complexity. The space requirement S(P) can be given as

S(P)=C + SP

Example

Statements	**s/e**	**Frequency**	**Total steps**
Algorithm sum(a,n)	0	-	0
{	0	-	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i]	1	n	n
return s;	1	1	1
}	0	-	0
			2n+3

The space requirement for algorithm given in example is S(P)= 2n+3. Neglect the constants

Space complexity is O(n)

Time Complexity	Space Complexity
Calculates time needed	Calculates memory space needed
Counts time for all statements.	Counts memory space for all variables even inputs and outputs
Depends mostly on input data size	Depends mostly on auxiliary variables size
More important for solution optimization	Less important with modern hardwares
Time complexity of merge sort is $\mathcal{O}(nlogn)$	Space complexity of merge sort is $\mathcal{O}(n)$

1.2 Asymptotic Notations

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity. Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time". There are mainly three asymptotic notations:

1. Big-O Notation (O-notation)

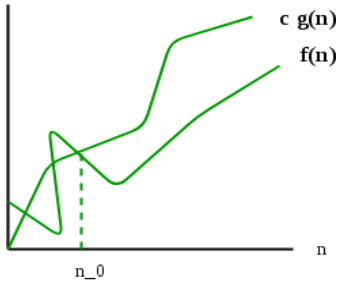
2. Omega Notation (Ω-Notation)

3. Theta Notation (Θ-Notation)

1. Big-O Notation (O-notation)

Big-oh notation denoted by „O" is a method of representing the upper bound of algorithms running time. Using big-oh notation we can give longest amount of time taken by the algorithm to complete.

Definition: The function f(n)=O(g(n)) iff there exist positive constants c and n_o such that $f(n) \leq cg(n)$ for all $n \geq n_0$

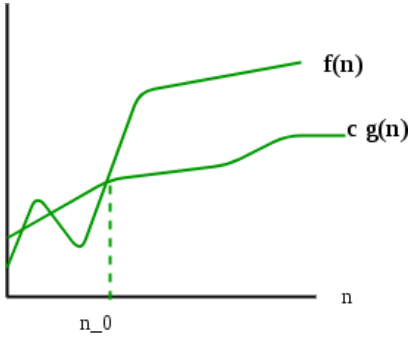


f(n) = O(g(n))

2. Omega Notation (Ω-Notation)

Omega notation denoted by Ω is a method of representing lower bound of algorithms running time. Using omega notation we can denote shortest amount of time taken by the algorithm to complete.

Definition: The function f(n)=Ω **(g(n)) iff there exist positive constants c and n_0 such that f(n) ≥c*g(n) for all n, n≥n_0**

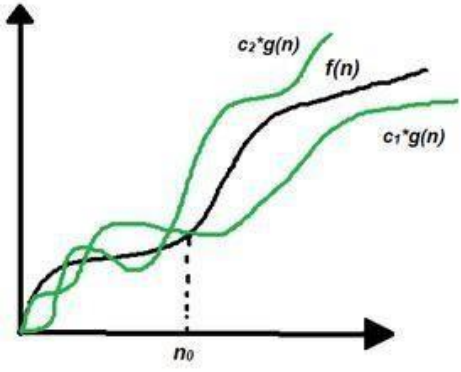


f(n) = Omega(g(n))

3. Theta Notation (Θ-Notation)

The Theta notation denoted as Θ is a method of representing running time between upper bound and lower bound.

Definition: The function f(n)= Θ(g(n)) iff there exist positive constants c1, c2 and n_0 such that **$c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$**



Linear Data structures - Time Complexities

Data Structure	Insert	Delete	Access	Search
Array	O(N)	O(N)	O(1)	O(N)
String	O(N)	O(N)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)	O(N)
Linked List	O(1), if inserted on head O(N), elsewhere	O(1), if deleted on head O(N), elsewhere	O(N)	O(N)

Searching Algorithms - Time Complexities

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)

Sorting Algorithms - Time Complexities

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)

1.3 AVL Trees –

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

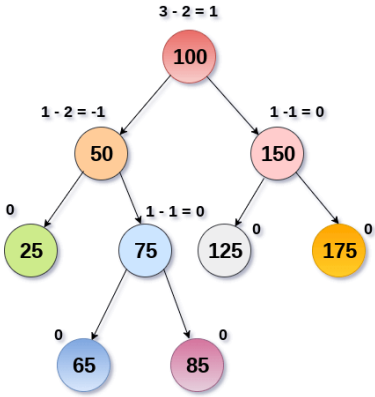
- AVL Trees are Self-Balanced Binary Search Trees.
- In AVL trees, the balancing factor of each node is either 0 or 1 or -1.
- Balance Factor of AVL Tree calculated as = Height of Left Sub-tree - Height of Right Sub-tree

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is **1,** it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is **-1,** it means that the left sub-tree is one level lower than the right sub-tree.
- If balance factor of any node is **0,** it means that the left sub-tree and right sub-tree contain equal height.



AVL Tree

Operations on an AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL Rotations

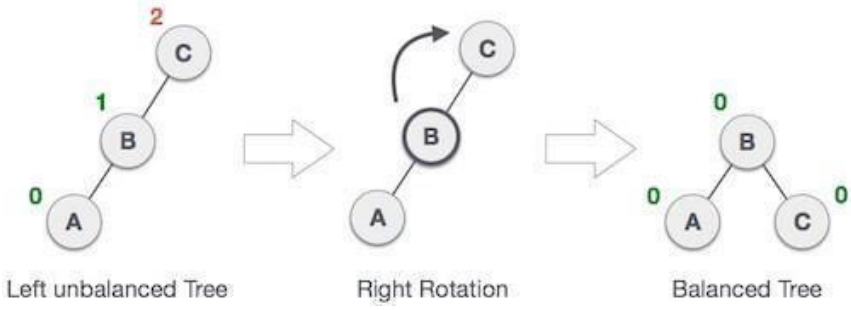
We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

The first two rotations **LL and RR are single rotations** and the next two rotations **LR and RL are double rotations**. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

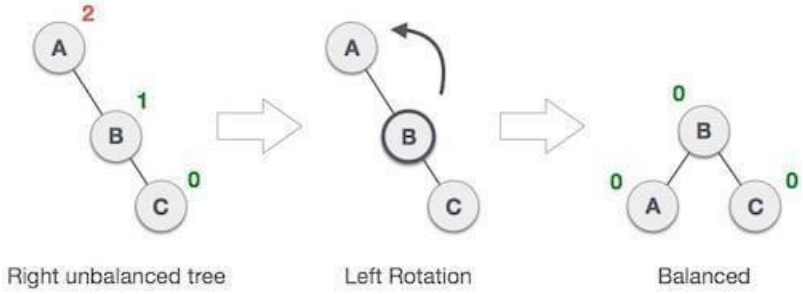
1. **LL Rotation**

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



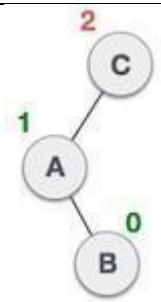
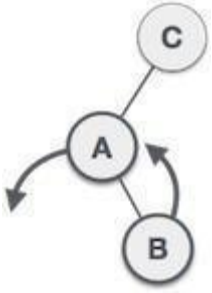
2. RR Rotation

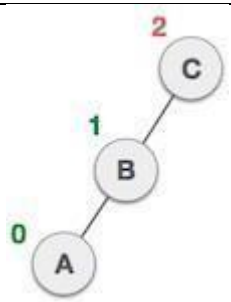
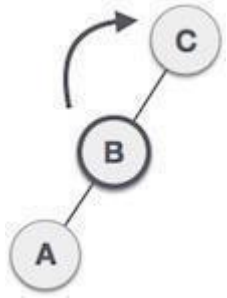
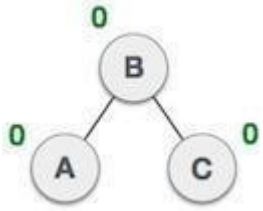
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

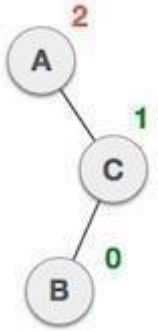
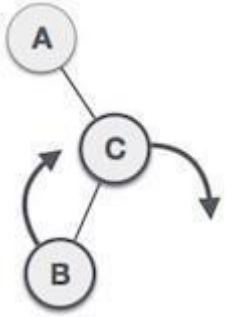
State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.

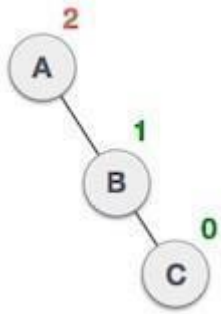
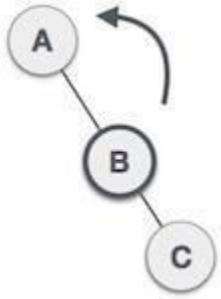
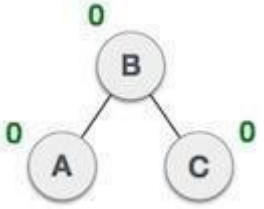
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on

full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

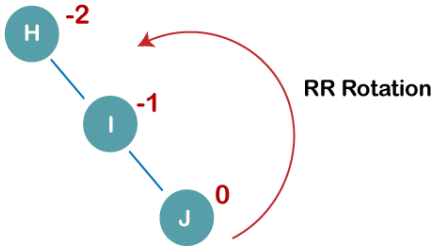
State	Action
	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.

	After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Q: Construct an AVL tree having the following elements

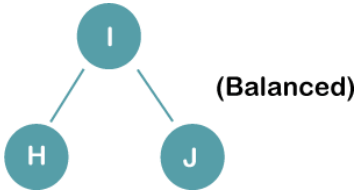
H, I, J, B, A, E

Sol: 1. Insert H, I, J

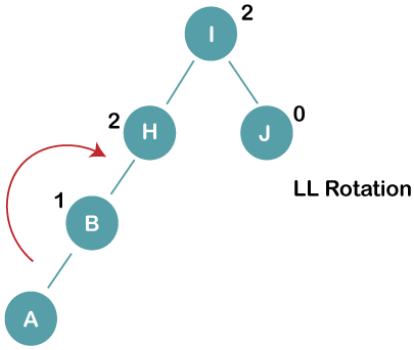


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H. The resultant balance tree is:

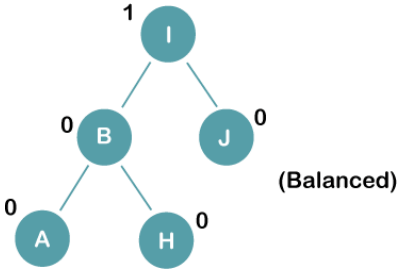
The resultant balance tree is:



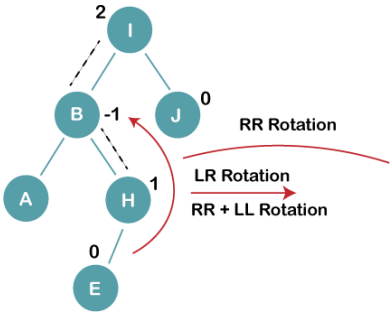
2. Insert B, A



On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H. The resultant balance tree is:



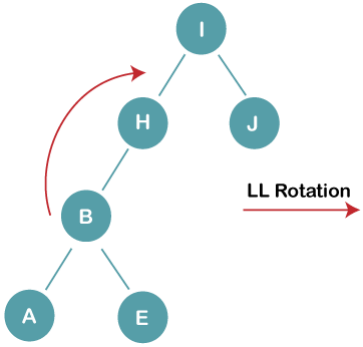
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

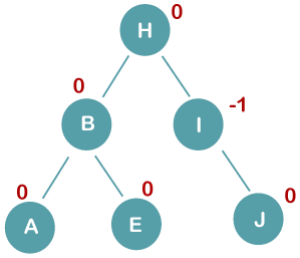
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:



3b) We first perform LL rotation on the node I

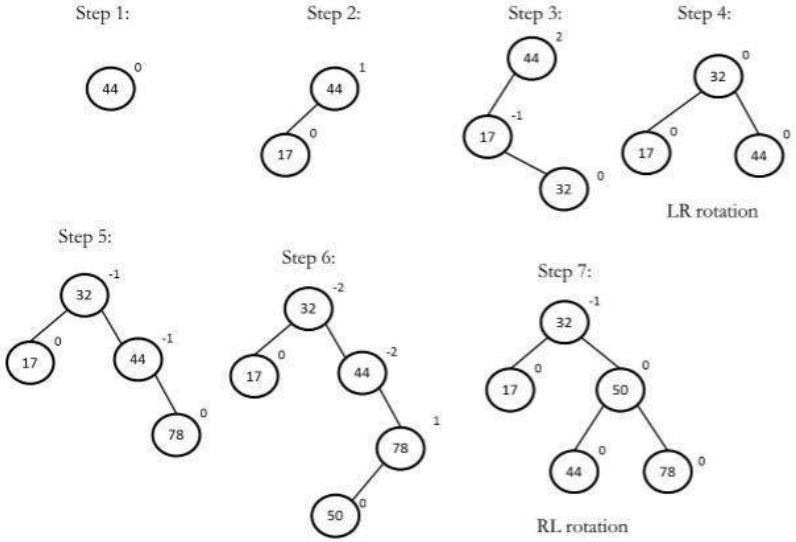
The resultant balanced tree after LL rotation is:

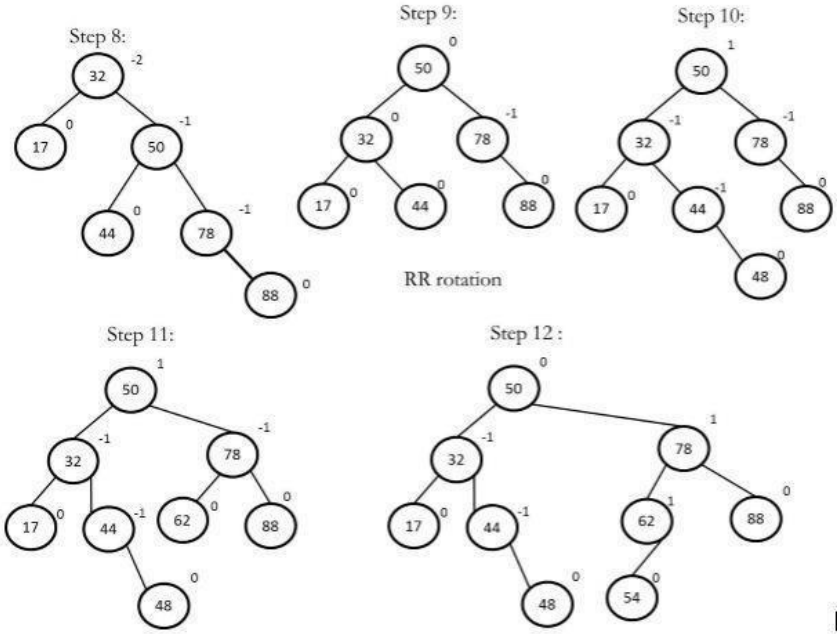


(Balanced)

Construct an AVL tree having the following elements

44, 17, 32, 78, 50, 88, 48, 62, 54

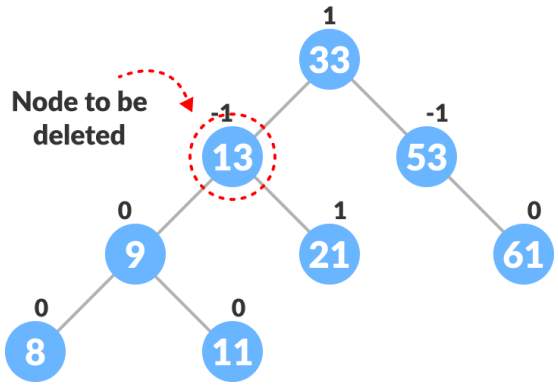




Delete a node

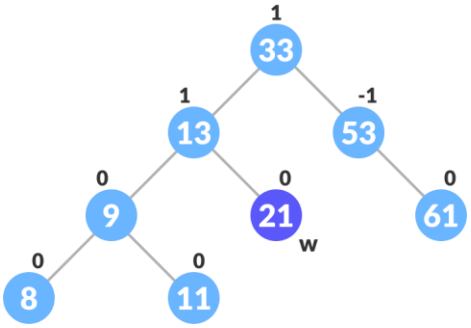
A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

1. Locate nodeToBeDeleted (recursion is used to find nodeToBeDeleted in the code used below).

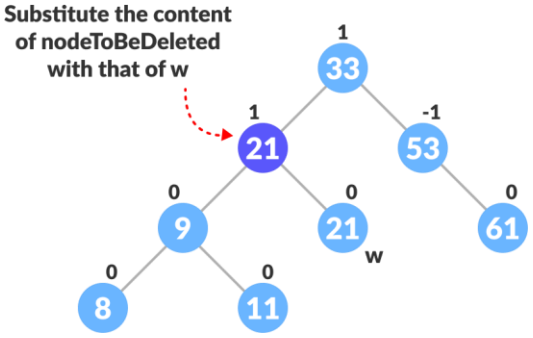


2. There are three cases for deleting a node:

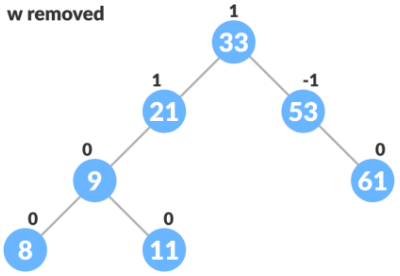
- If nodeToBeDeleted is the leaf node (ie. does not have any child), then remove nodeToBeDeleted.
- If nodeToBeDeleted has one child, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.
- If nodeToBeDeleted has two children, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).



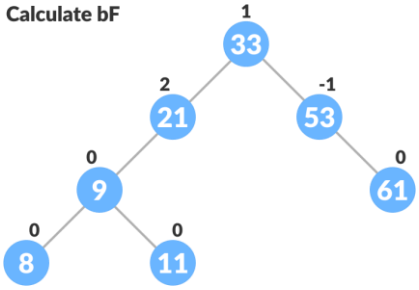
a) Substitute the contents of nodeToBeDeleted with that of w



b) Remove the leaf node w.



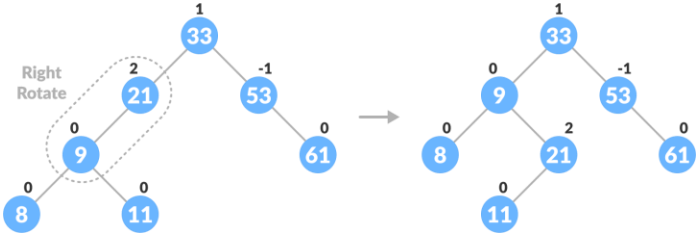
3. Update balanceFactor of the nodes.



4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

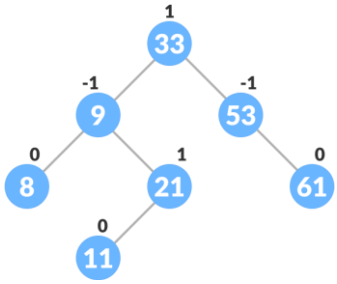
a) If balanceFactor of currentNode > 1,

b) If balanceFactor of leftChild >= 0, do right rotation.



- Else do left-right rotation.
- If balanceFactor of currentNode < -1,
- If balanceFactor of rightChild <= 0, do left rotation.
- Else do right-left rotation.

5. The final tree is:



Applications

1. Databases:

AVL trees are used in databases to ensure quick data retrieval. Their self-balancing property makes them ideal for implementing index structures that allow for efficient searching, insertion, and deletion operations.

2. File Systems:

File systems often use AVL trees to manage directories and files. The balance of the tree ensures that file operations are performed efficiently, which is critical for maintaining the performance of the file system.

3. Memory Management:

AVL trees are used in memory management systems to keep track of free memory blocks. The balanced nature of AVL trees ensures quick allocation and deallocation of memory.

4. Networking:

In networking, AVL trees can be used to manage routing tables efficiently. The balanced structure helps in maintaining the routing information for quick lookup, insertion, and deletion.

5. Compilers:

Compilers use AVL trees to implement symbol tables, which store information about variables, functions, and other entities. The efficient searching capability of AVL trees helps in quick symbol resolution during compilation.

6. Geographical Information Systems (GIS):

AVL trees can be used in GIS for indexing spatial data. The balanced structure ensures efficient query processing for operations such as range searches and nearest neighbor searches.

7. Telecommunication:

In telecommunications, AVL trees can be used to manage call routing and connection management. The efficient lookup and update operations help in maintaining real-time performance.

8. Scheduling Systems:

AVL trees are used in various scheduling systems to keep track of tasks and resources. The balanced nature ensures that scheduling operations are performed quickly and efficiently.

9. Spell Checkers:

AVL trees can be used in spell checkers to store dictionaries. The efficient searching capability helps in quickly finding words and suggesting corrections.

10. Gaming:

In gaming, AVL trees can be used to manage objects and characters in a game world. The balanced structure helps in efficient collision detection, scene management, and rendering.

11. Priority Queues:

AVL trees can be used to implement priority queues. The balanced nature ensures that operations such as insert, delete, and find-minimum can be performed efficiently.

12. Version Control Systems:

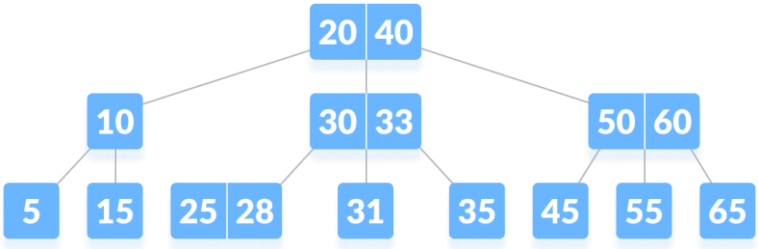
In version control systems, AVL trees can be used to manage versions of files and directories. The efficient

update and retrieval operations help in maintaining and accessing different versions quickly.

1.4 B-Trees

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.

It is also known as a height-balanced m-way tree.



Why do you need a B-tree data structure?

The need for B-tree arose with the rise in the need for lesser time in accessing physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk access.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large, and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Basic Operations of B Trees

The operations supported in B trees are Insertion, deletion and searching with the time complexity of O(log n) for every operation.

Insertion operation

The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure −

Step 1 − Calculate the maximum (m−1)

and, minimum (⌈m/2⌉−1)

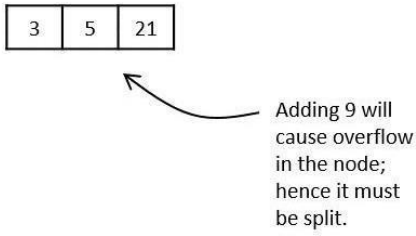
number of keys a node can hold, where m is denoted by the order of the B Tree.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

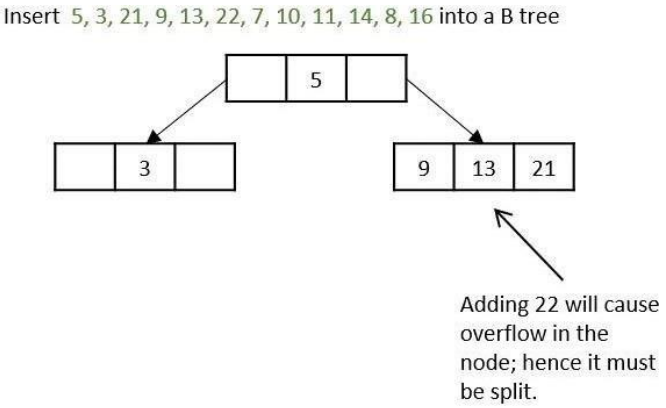
- Order (m) = 4
- Maximum Keys (m – 1) = 3
- Minimum Keys ($\left\lceil \frac{m}{2} \right\rceil$) – 1 = 1
- Maximum Children = 4
- Minimum Children ($\left\lceil \frac{m}{2} \right\rceil$) = 2

Step 2 − The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children.

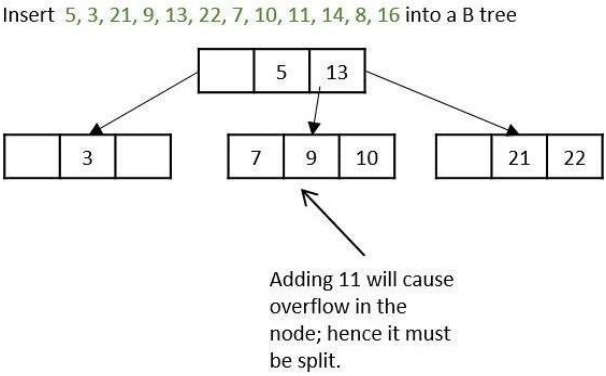
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



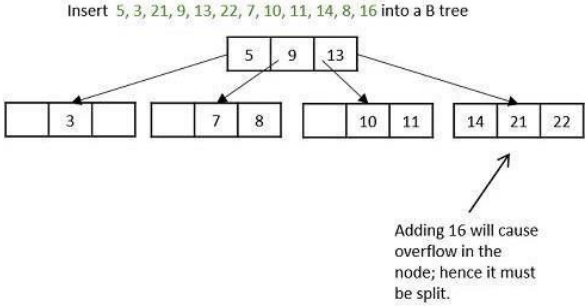
Step 3 − All the leaf nodes must be on the same level.



The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.

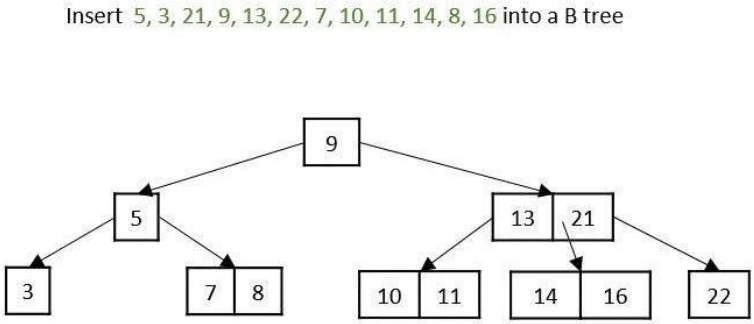


Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.



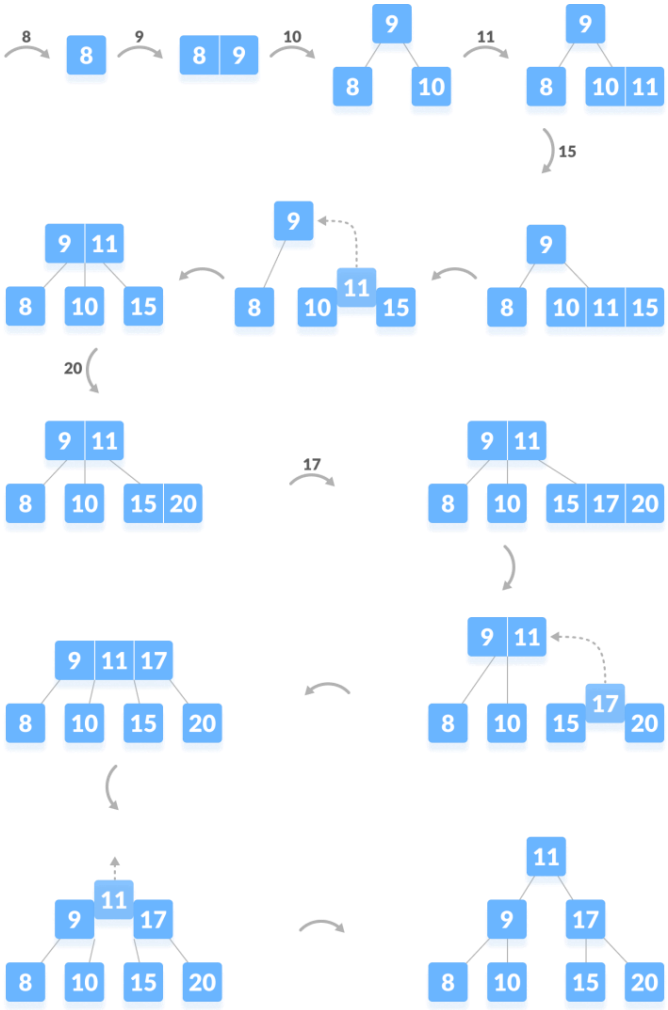
While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key

becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.



The final B tree after inserting all the elements is achieved.

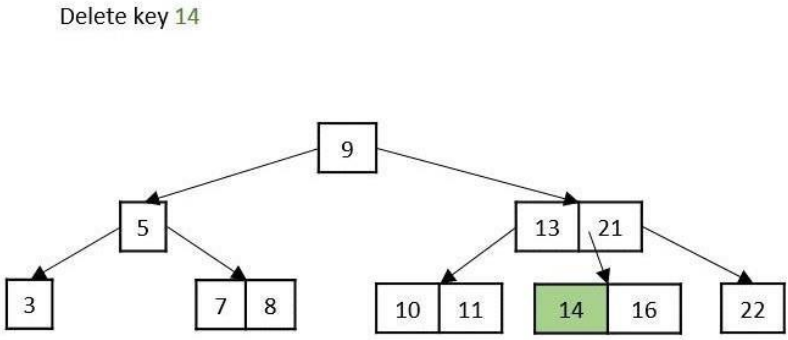
Example 2: The elements to be inserted are 8, 9, 10, 11, 15, 20, 17

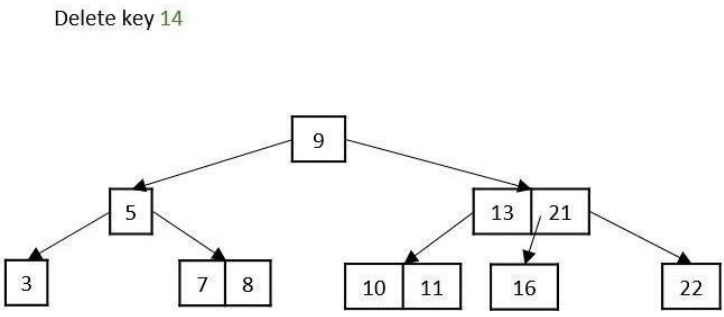

8
9
10
11
15
20
17

Deletion operation

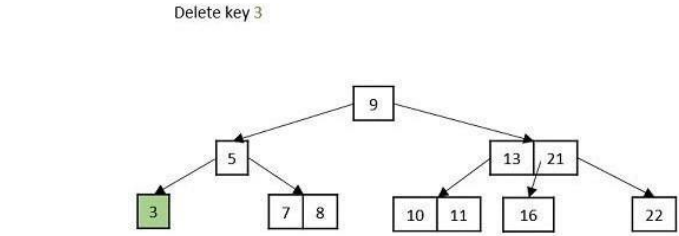
The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows −

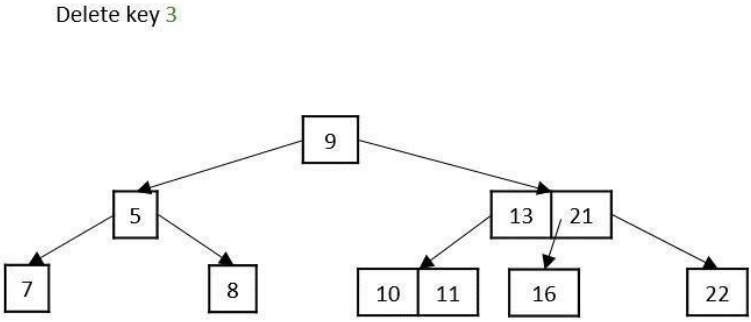
Case 1 − If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.



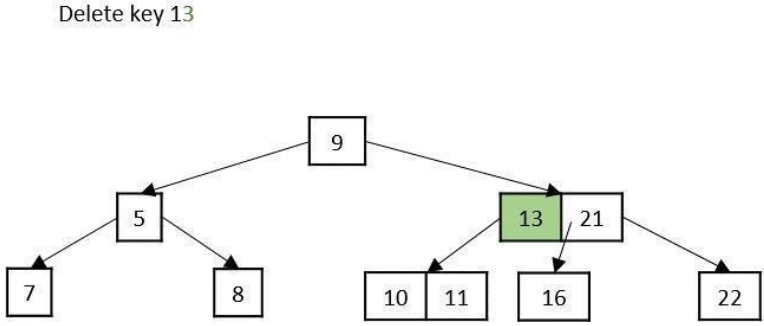


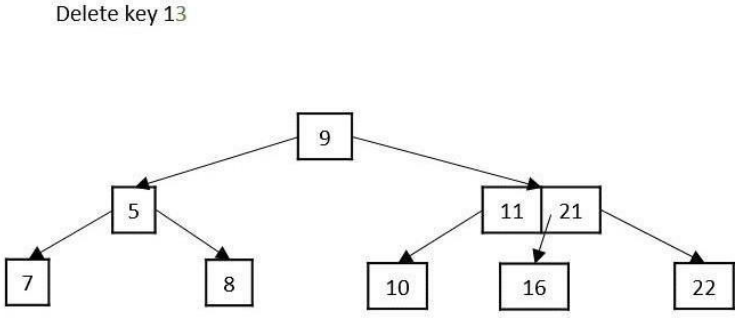
Case 2 − If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them.



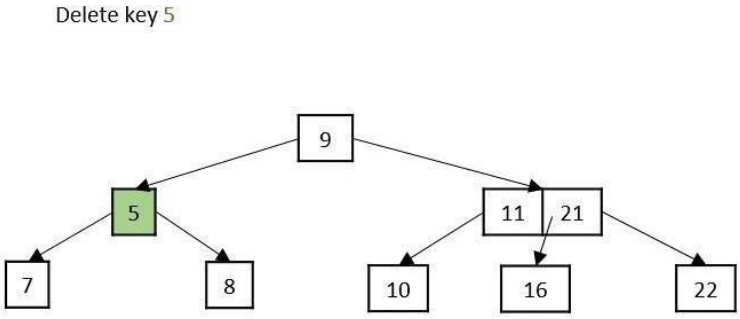


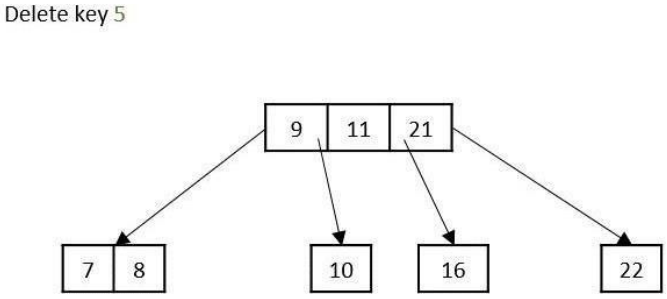
Case 3 − If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, they‟re merged together.





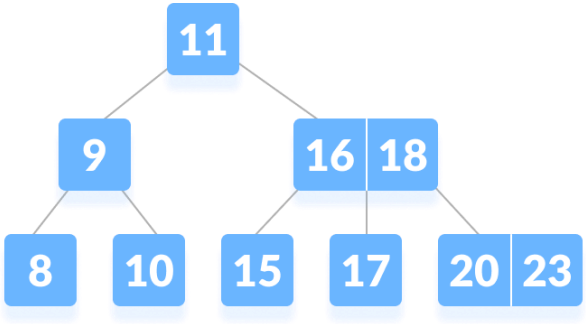
Case 4 − If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent.



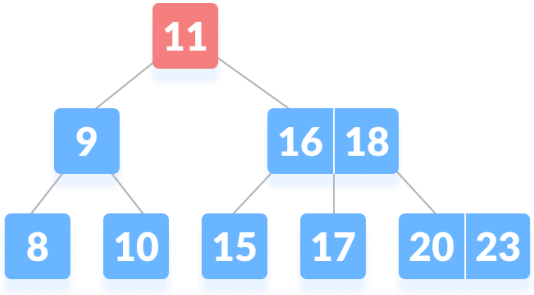


Searching an element in a B-tree

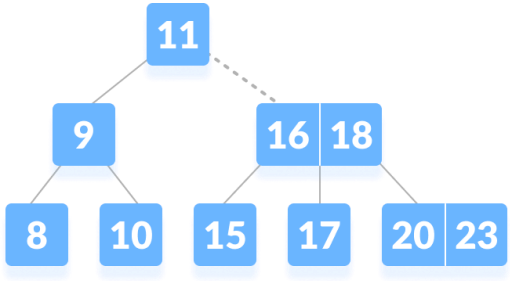
a) Let us search key k = 17 in the tree below of degree 3



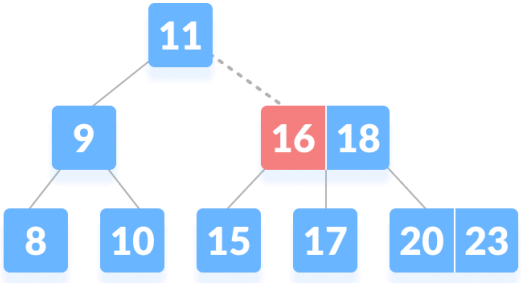
b) k is not found in the root so, compare it with the root key.



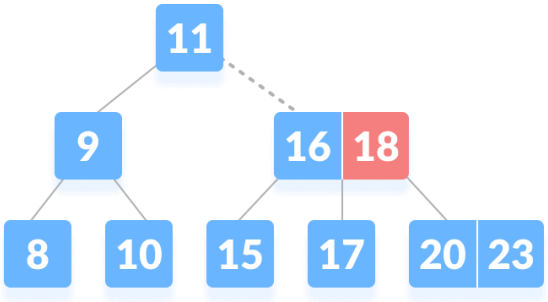
C) Since k > 11, go to the right child of the root node.



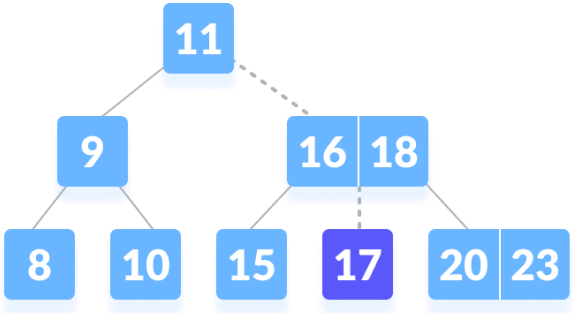
d) Compare k with 16. Since k > 16, compare k with the next key 18.



E) Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



f) k is found.



Applications

- Large databases employ it to access information stored on discs.
- Using the B-Tree, finding data in a data set can be done in a great deal less time.

- Multilevel indexing is possible with the indexing feature.
- The B-tree method is also used by the majority of servers.
- In CAD systems, B-Trees are used to catalogue and search geometric data.
- Other applications of B-Trees include encryption, computer networks, and natural language processing.
- Since accessing values stored in a large database that is stored on a disc takes a long time, B trees are used to index the data and provide quick access to the actual data stored on the disks.
- In the worst case, it takes O(n) running time to search a database with n key values that is not sorted or indexed. However, if we use B Tree to index this database, it will be searched in O (log n) time in worst case.

Short Answer Questions

1. Define an algorithm and list any two characteristics of an algorithm.
2. State any four applications of algorithms.
3. Differentiate between time complexity and space complexity.
4. Define Big-O notation with an example.
5. What is the balance factor of an AVL tree?
6. List the four types of AVL rotations.
7. What is a B-Tree? Mention any two advantages of B-Trees.
8. State the time complexities of search, insertion, and deletion in a B-Tree.
9. Mention any two applications of AVL Trees.
10. Mention any two applications of B-Trees.

Long Answer Questions

1. Explain the properties of an algorithm and discuss the process involved in algorithm design and analysis with suitable examples.
2. Explain time complexity and space complexity with suitable examples.
3. Explain Big-O, Omega (Ω), and Theta (Θ) asymptotic notations with suitable examples.
4. Construct an AVL Tree by inserting the following elements:
   H, I, J, B, A, E.
   Show every rotation performed during insertion.
5. Construct an AVL Tree for the following elements:
   44, 17, 32, 78, 50, 88, 48, 62, 54.
   Illustrate all necessary rotations.
6. Explain the insertion operation in a B-Tree with a suitable example.
7. Analyze the deletion operation in an AVL Tree and explain how different rotations restore balance after deletion.
8. Explain the deletion operation in a B-Tree by discussing all possible cases with suitable examples.

Case Study – Unit I

Designing an Efficient Student Information Management System Using AVL Trees and B-Trees

Course

Algorithms, Data Structures & Algorithm Analysis (ADS&AA)

Difficulty Level

Intermediate

Learning Outcomes

After completing this case study, students will be able to:

- Analyze algorithm efficiency using time and space complexity.
- Apply Big-O, Omega, and Theta notations.
- Compare Binary Search Trees, AVL Trees, and B-Trees.
- Design efficient data structures for large-scale applications.
- Recommend suitable algorithms for real-world scenarios.

Background

ABC Engineering College maintains academic records for more than **25,000 students**.

The database stores:

- Student ID
- Name
- Branch
- Semester
- CGPA

- Attendance
- Examination Results

Every day thousands of operations are performed:

- Student search
- Record insertion
- Record deletion
- Updating marks
- Generating reports

Initially, the college stored all student records in a **Binary Search Tree (BST)**.

As the number of records increased, the BST became highly unbalanced, causing searches to become slow.

The college management decided to redesign the database using efficient data structures.

Problem Statement

The college wants to develop a system that can:

- Search student records quickly
- Insert new admissions efficiently
- Delete graduated students
- Minimize memory usage
- Reduce disk access time
- Handle thousands of operations every minute

Existing System

Data Structure Used

Binary Search Tree

Problems

- Tree becomes skewed.
- Search time increases.
- Insertion becomes slow.
- Deletion affects performance.
- Database response becomes poor.

Worst-case complexity

Search = O(n)

Insert = O(n)

Delete = O(n)

Proposed Solution

The system designer recommends two different data structures.

Part A – AVL Tree

AVL Tree is selected for storing records in the application memory because it automatically balances itself after every insertion or deletion. Balance is maintained using rotations (LL, RR, LR, RL), ensuring efficient operations.

Advantages

- Balanced tree
- Fast searching
- Efficient insertion
- Efficient deletion
- Suitable for frequently updated data

Time Complexity

- Search → O(log n)
- Insert → O(log n)
- Delete → O(log n)

Part B – B-Tree

Student records stored permanently on disk are indexed using a B-Tree. Since a B-Tree stores multiple keys per node, it reduces the height of the tree and minimizes disk accesses, making it ideal for databases.

Advantages

- Reduced disk access
- Faster indexing
- Suitable for databases
- Supports millions of records
- Balanced automatically

Time Complexity

- Search → O(log n)
- Insert → O(log n)
- Delete → O(log n)

Algorithm Analysis

Searching

Suppose the database contains

25,000 student records

Data Structure	Complexity	Approximate Performance
Linear Search	O(n)	Slow
Binary Search Tree	O(n) (Worst Case)	Moderate
AVL Tree	O(log n)	Fast
B-Tree	O(log n)	Very Fast

Space Complexity

Additional memory required

- BST → Low
- AVL Tree → Slightly higher (stores balance factor)
- B-Tree → Efficient disk block utilization

Asymptotic Analysis

Best Case

Ω(log n)

Searching the balanced tree.

Average Case

Θ(log n)

Balanced insertions and searches.

Worst Case

O(log n)

AVL Tree and B-Tree remain balanced.

Real-World Applications

The same concepts are used in:

- Banking Systems
- Library Management
- Hospital Information Systems
- Railway Reservation Systems
- E-Commerce Websites
- Cloud Databases
- University ERP Systems

Discussion Questions

1. Why does an ordinary Binary Search Tree become inefficient for large datasets?
2. Explain why AVL Trees provide faster searching than BSTs.
3. Why are B-Trees preferred in database management systems?
4. Compare AVL Tree and B-Tree in terms of storage and performance.
5. Which asymptotic notation best represents the worst-case performance of AVL Tree search?
6. How do AVL rotations maintain tree balance?
7. Why is minimizing disk access important in database systems?

---

8. Suggest another real-world application where AVL Trees can be effectively used.

Conclusion

By replacing the Binary Search Tree with an **AVL Tree** for in-memory operations and a **B-Tree** for disk-based indexing, the college significantly improves search speed, insertion and deletion efficiency, and scalability. Algorithm analysis using **time complexity**, **space complexity**, and **asymptotic notations** demonstrates that balanced trees provide predictable **O(log n)** performance, making them suitable for large-scale academic information systems.

UNIT-II

Heap Trees (Priority Queues): Min and Max Heaps
Operations and Applications.
Graphs: Terminology,
Representations,
Basic Search and Traversals,
Connected Components and Bi connected Components,
applications.
Divide and Conquer: The General Method,
Quick Sort,
Merge Sort,
Strassen's matrix multiplication,
Convex Hull.

Heap

The heap data structure is a complete binary tree where each node of the tree has an orderly relationship with its successors. Binary search trees are totally ordered, but the heap data structure is only partially ordered. It is suitable for inserting and deleting minimum value operations.

Heap is an array object that is considered as a complete binary tree. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled at all levels except possibly the lowest, which is filled from the left upwards to a point. Heap data structures are suitable for implementing priority queues. The heap serves as a foundation of a theoretically important sorting algorithm called heap sort, which we will discuss after defining the heap.

A heap can be used as a priority queue: the highest priority item is at the root and is trivially extracted. But if the root is deleted, you are left with two sub-trees and you must efficiently re- create a single tree with the heap property.

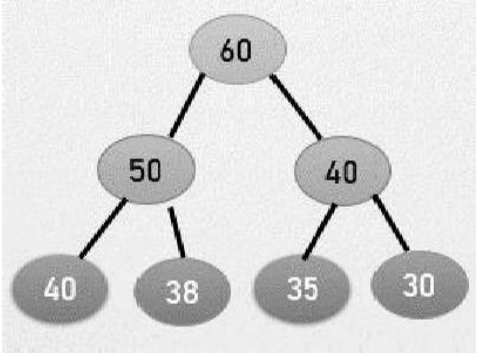
The value of the heap structure is that you can both extract the highest priority item and insert a new one in O(logn) time.

A heap can be defined as binary trees with keys assigned to its nodes (one key per node). The two types of heaps are:

1. Max heap
2. Min heap

Max heap

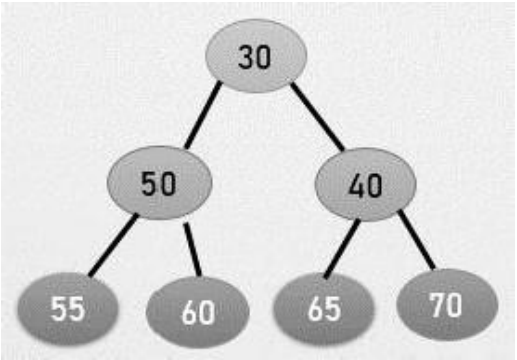
The key present at the root node must be greatest or equal to the keys present at all of its children. The same property must be true for all sub-trees in that Binary Tree.



Min heap

The key present at the root node must be minimum or equal to the keys present at all of its children.

The same property must be true for all sub-trees in that Binary Tree.



Heap Tree construction

Create a new node at the end of heap.

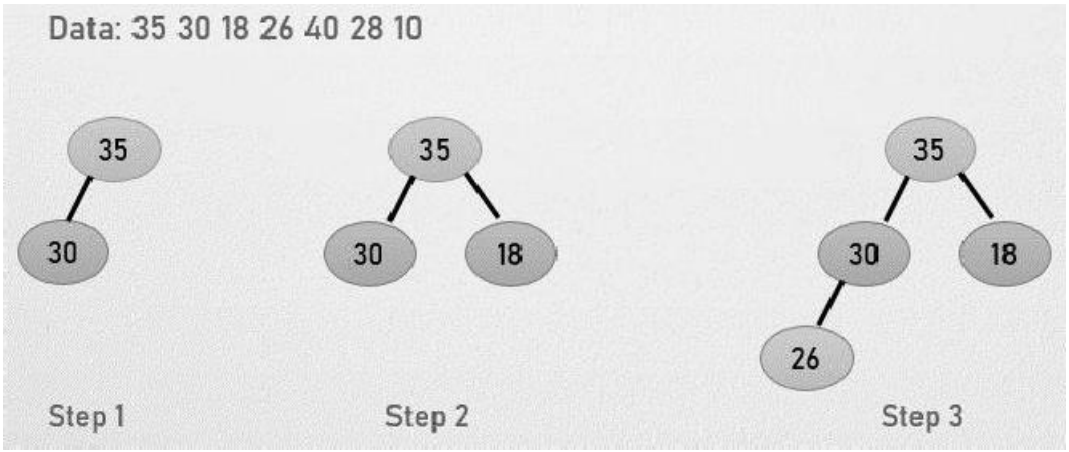
Assign new value to the node.

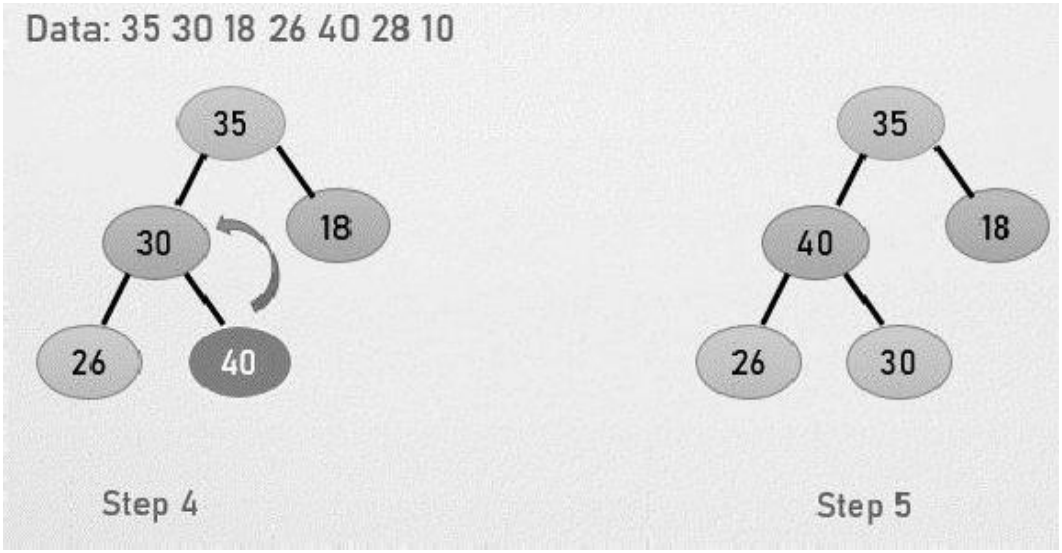
Compare the value of this child node with its parent.

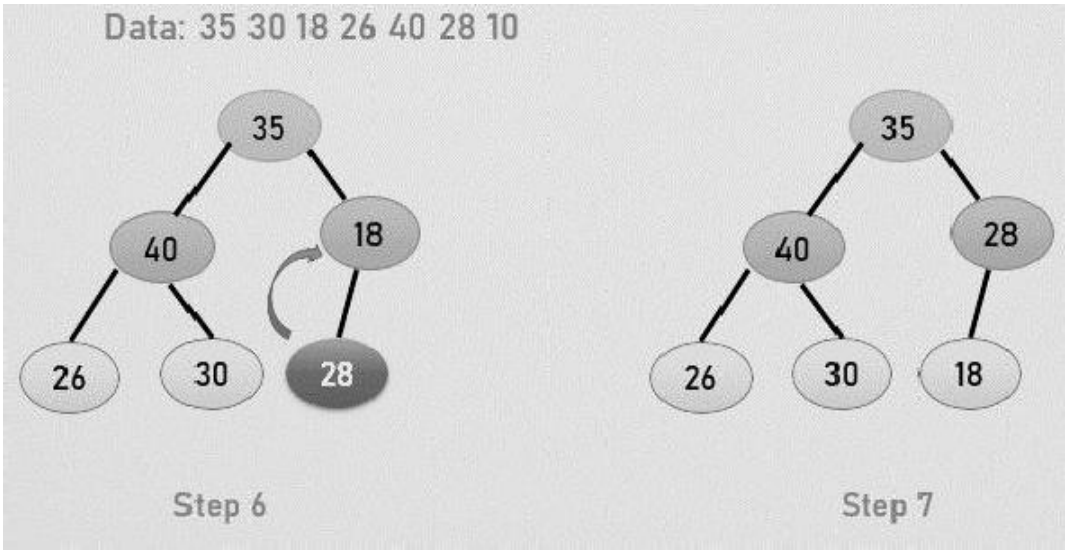
If value of parent is less than child, then swap them.

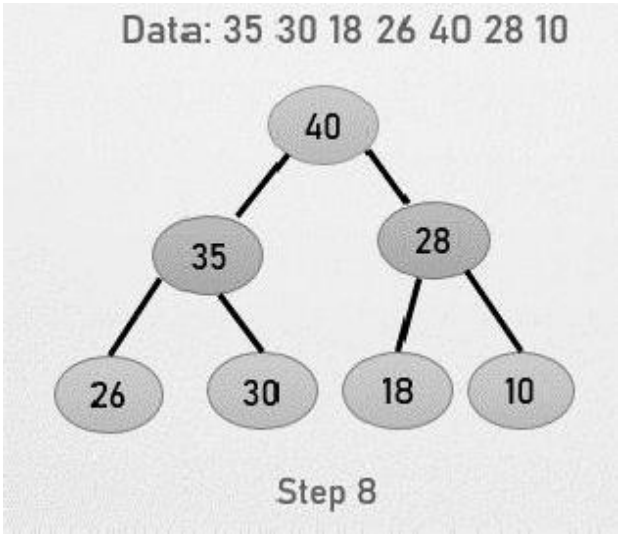
Repeat step 3 & 4 until Heap property holds.

Example: Max heap



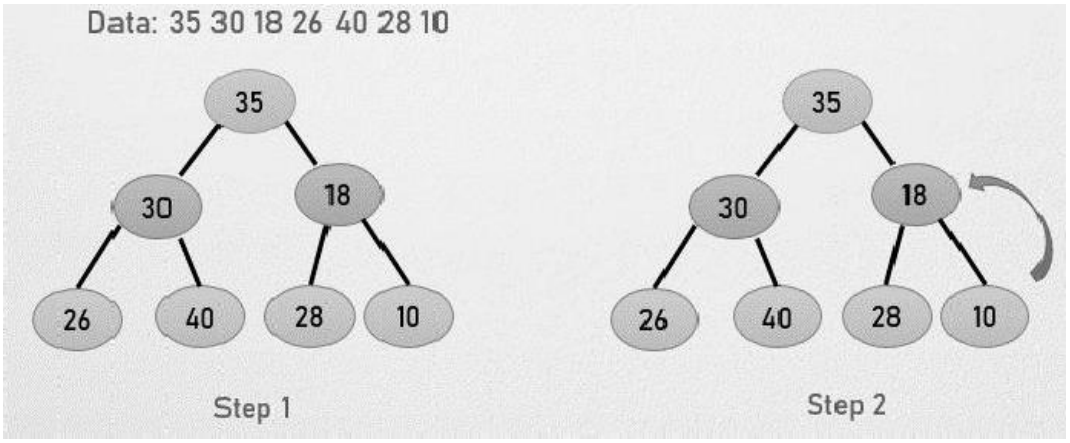

Data: 35 30 18 26 40 28 10
35
30
18
26
40
Step 4
35
40
18
26
30
Step 5

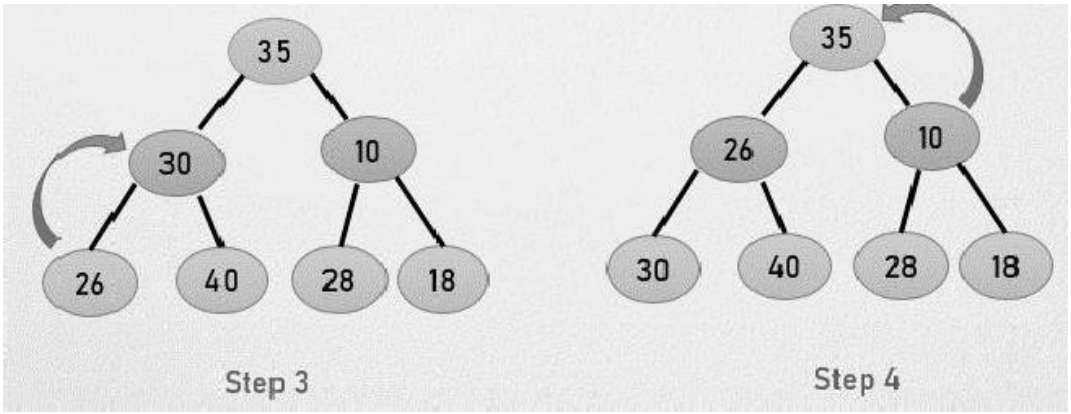

Data: 35 30 18 26 40 28 10
35
40
18
26
30
28
Step 6
35
40
28
26
30
18
Step 7

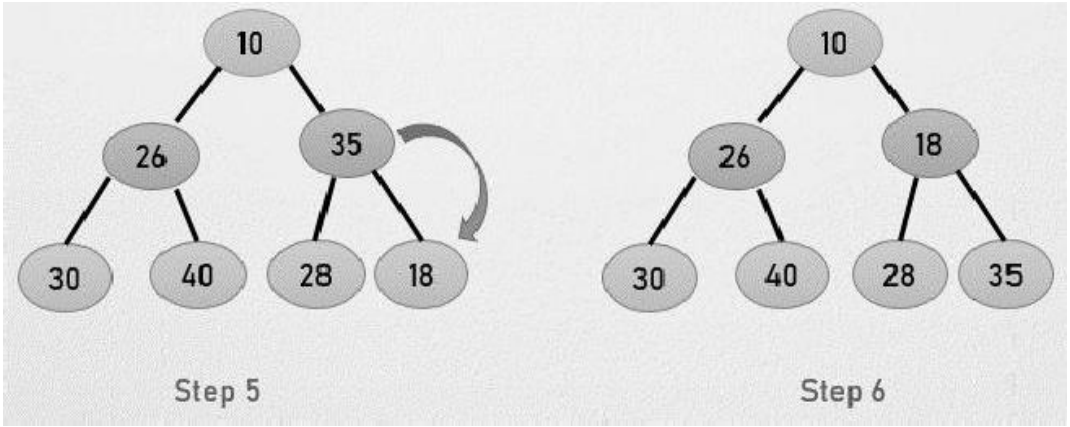

Data: 35 30 18 26 40 28 10
40
35
28
26
30
18
10
Step 8

Heapify Method (Min Heap)

Complexity: O(n)







Deletion Operation on Heap Tree

There are two methods for deletion in heap tree.

Method 1

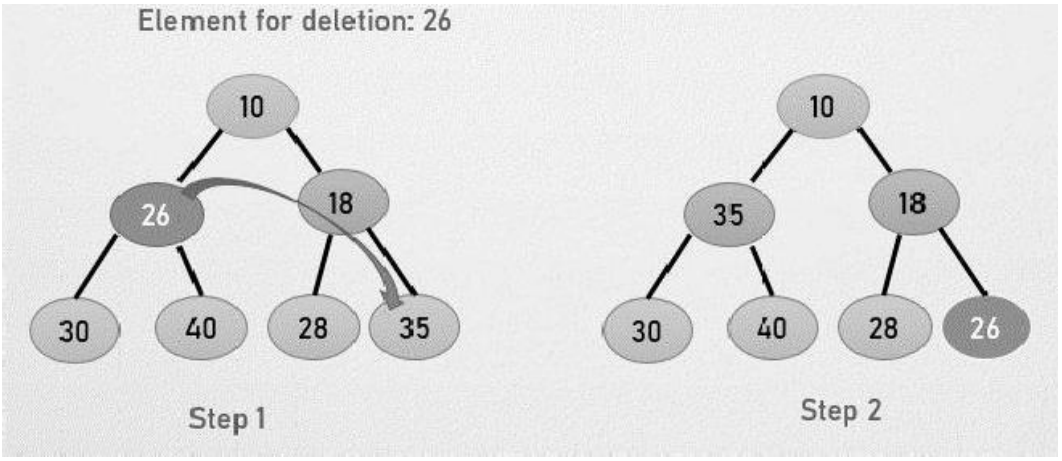
1. Remove root node.
2. Move the last element of last level to root.
3. Compare the value of this child node with its parent.
4. If value of parent is less than child, then swap them.
5. Repeat step 3 & 4 until Heap property holds.

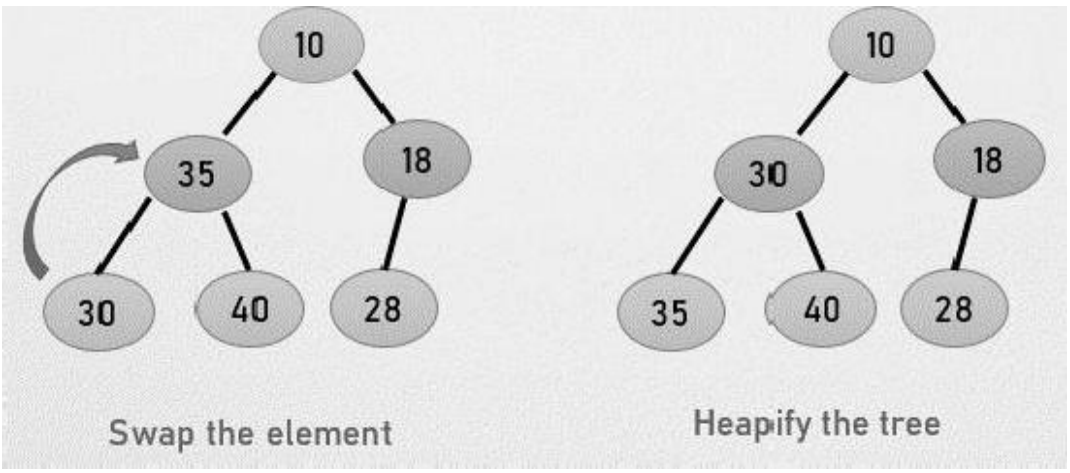
Method 2

1. Select the element to be deleted.
2. Swap it with the last element.
3. Remove the last element.
4. Heapify the tree.

Example:

Deletion operation on heap





Applications of Heaps

Priority queue implementation

Heap sort

Priority Queue

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

Every element in a priority queue has some priority associated with it.

An element with the higher priority will be deleted before the deletion of the lesser priority.

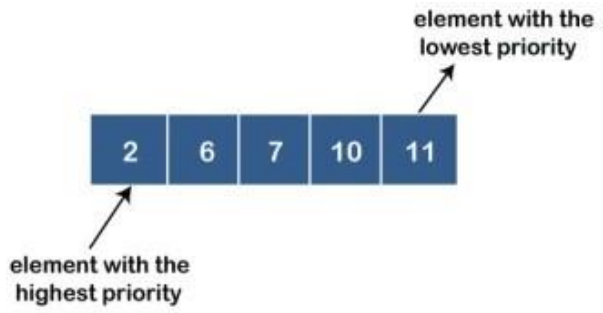
If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Types of Priority Queue

There are two types of priority queue

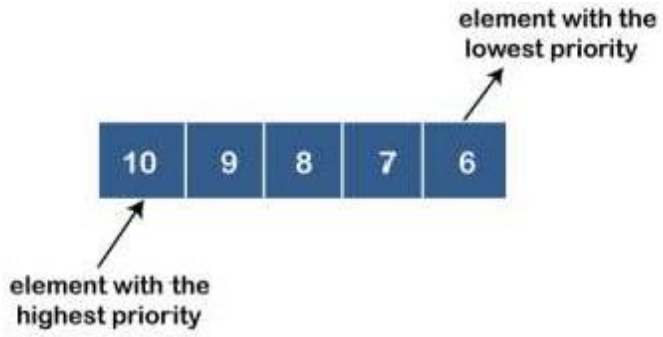
Ascending order priority queue:

In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



Descending order priority queue

In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Priority Queue Operations

Insert

Delete

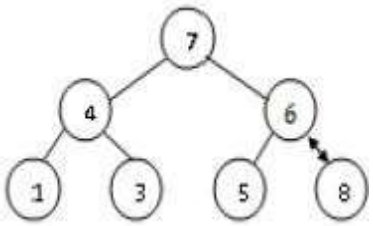
Peeking from the Priority Queue

Extract-Max/Min from the priority Queue.

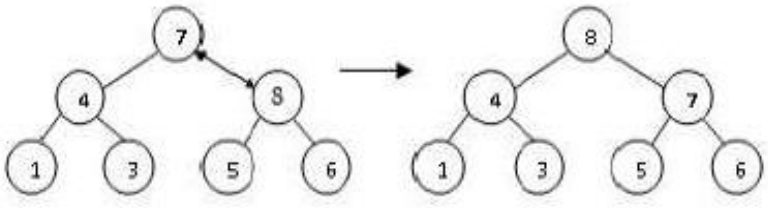
Insertion operation

To insert a new key into a heap, add a new node with key K after the last leaf of the existing heap.

Then, shift K up to its suitable place in the new heap. Consider inserting value 8 into the heap shown in the figure



Compare 8 with its parent key. Stop if the parent key is greater than or equal to 8. Else, swap these two keys and compare 8 with its new parent (Refer to figure 14.8). This swapping continues until 8 is not greater than its last parent or it reaches the root. In this algorithm too, we can shift up an empty node until it reaches its proper position, where it acquires the value 8.



This insertion operation does not require more key comparisons than the heap's height. Since the height of a heap with n nodes is about log2n, the time efficiency of insertion is in O(log n).

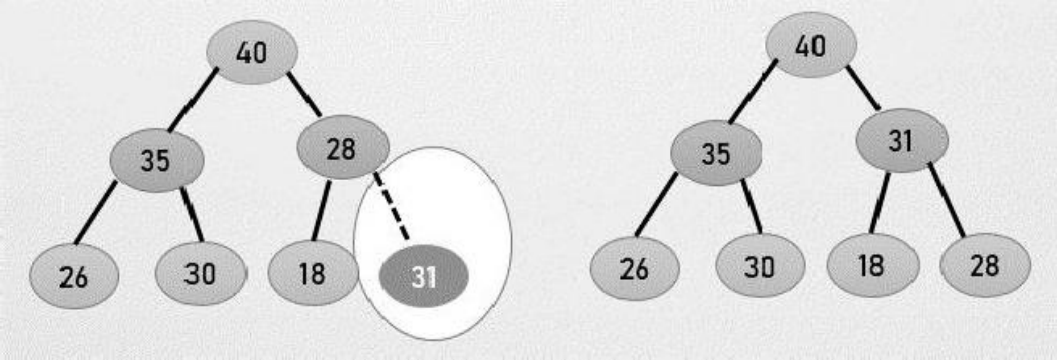
Insertion operation algorithm

If there is no node, create a newNode.

else (a node is already present)

insert the newNode at the end (last node from left to right.)

Heapify the array



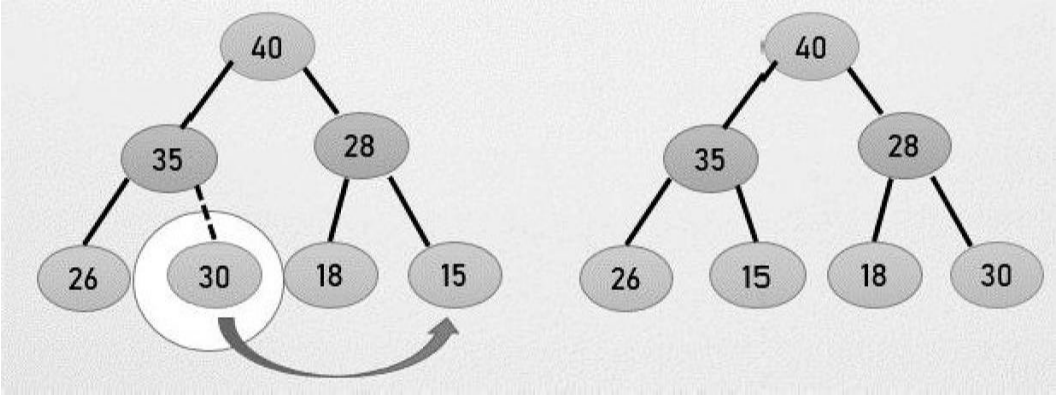
Delete operation

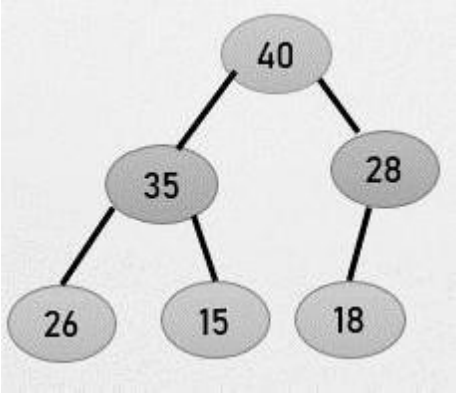
If node To Be Deleted is the leaf Node remove the node

Else swap node To Be Deleted with the last Leaf Node

remove noteToBeDeleted

heapify the array

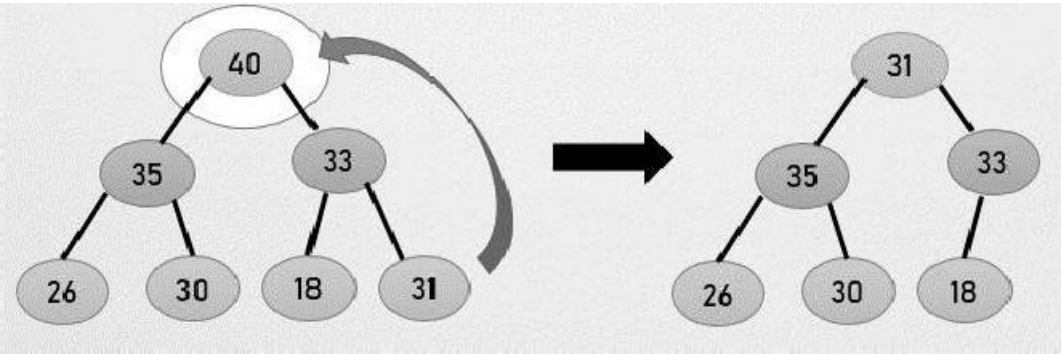




Deleting the Root of a Heap/ Extract Maximum

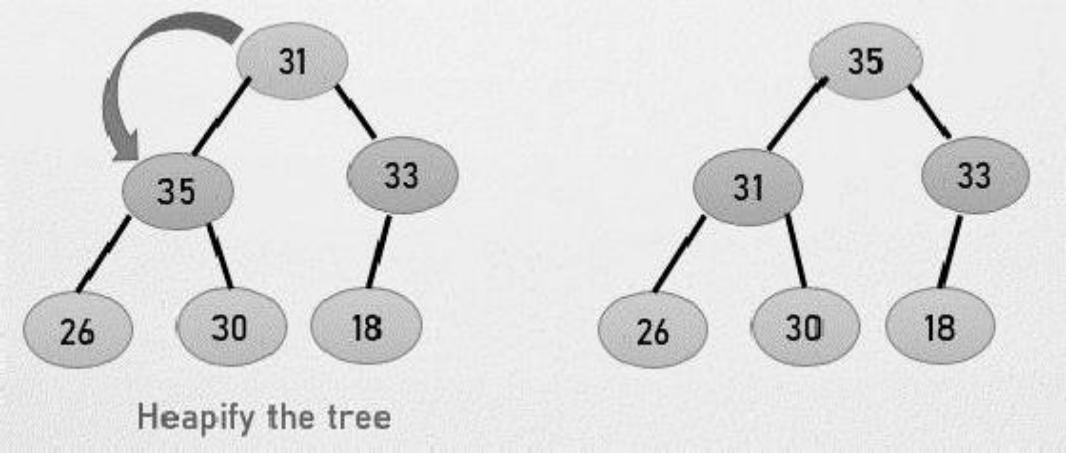
The following steps show the method to delete the root key from a heap in the figure.

Step 1: Exchange the root's key with the last key K of the heaps as shown in the figure.



Step 2: Decrease the heap's size by 1

Step 3: "Heapify" the smaller tree by shifting K down the tree as we did in the bottom-up heap construction algorithm. That is, verify the parental dominance for K and if it holds, we complete the process. If not, swap K with the largest of its children and repeat this operation until the parental dominance condition holds for K in its new position.



We can determine the efficiency of deletion by the number of key comparisons required to "heapify" the tree after the swap is done, and the size of the tree is decreased by 1. Since it does not need more key comparisons than twice the heap's height, the time efficiency of deletion is in O(log n).

Heap Sort

Heap sort is sorting technique based upon Binary Heap data structure.It is comparison-based sorting technique.It processes the elements by creating the min heap or max heap using the elements of the array. Heap sort basically recursively performs two main operations -

Build a heap H, using the elements of array.

Repeatedly delete the root element of the heap formed in 1st phase.

Steps for Heap Sort

Construct a Binary Tree from the list of Elements.

Transform the Binary Tree into Max Heap / Min Heap.

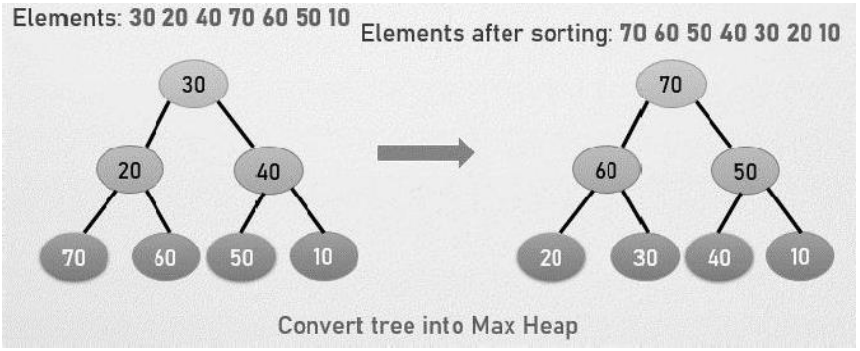
Delete the root element from Max Heap / Min Heap

Reducing the size of heap by 1

Heapify the root of the tree.

Put the deleted element into the Sorted list.

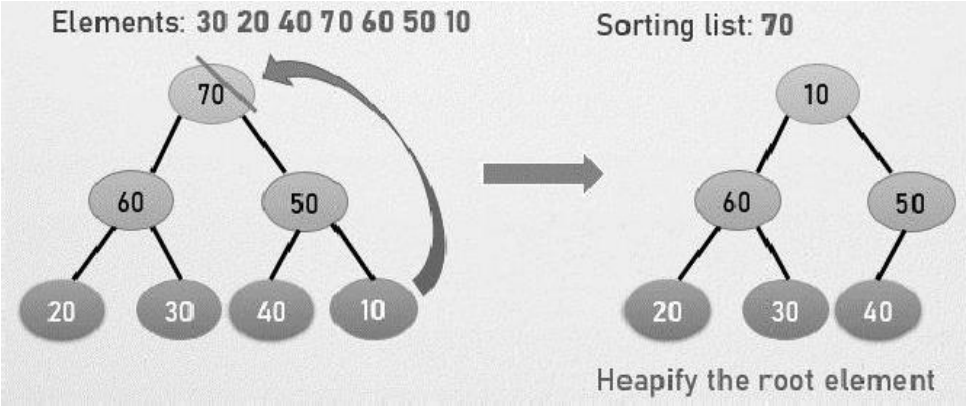
Repeat the same until Min Heap becomes empty.



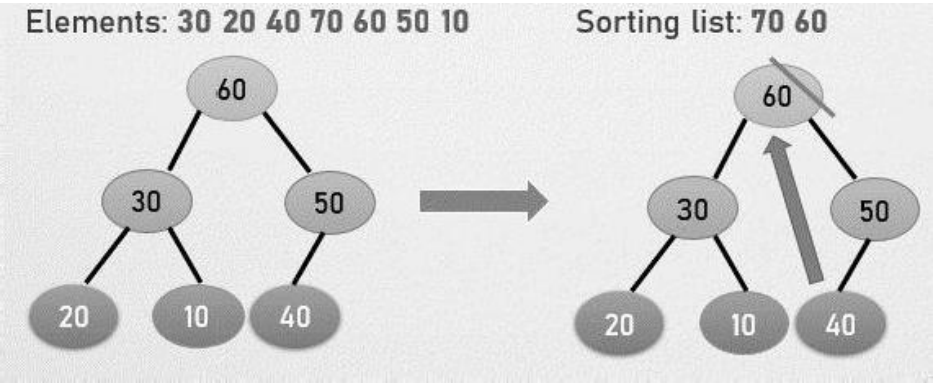
Heap sort first converts the initial array into a heap. The heapsort algorithm uses 'heapify' method to complete the task. The heapify algorithm, as given in the above code, receives a binary tree as input and converts it to a heap. Then, the root is compared with its two children, and the larger child is swapped with it. This may result in one of the left or right sub-trees losing the heap property. As a result, the heapify algorithm is recursively applied to the suitable sub-tree rooted at the node whose value was swapped with the root. This process continues until a leaf node is reached, or until the heap property is satisfied in the sub-tree.

Example Heap sort:

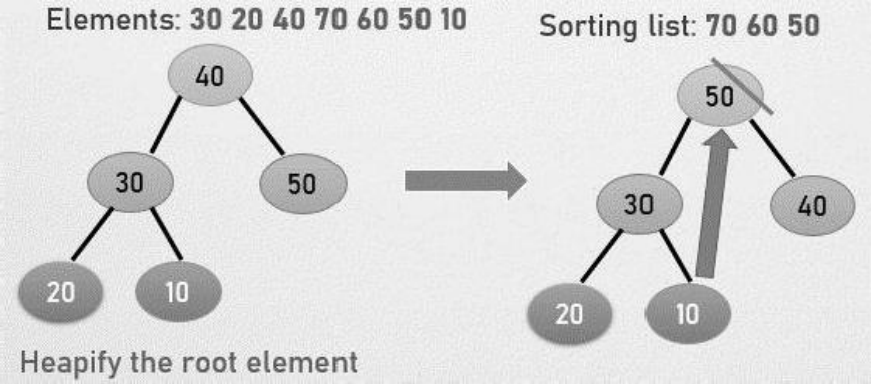
Step 1



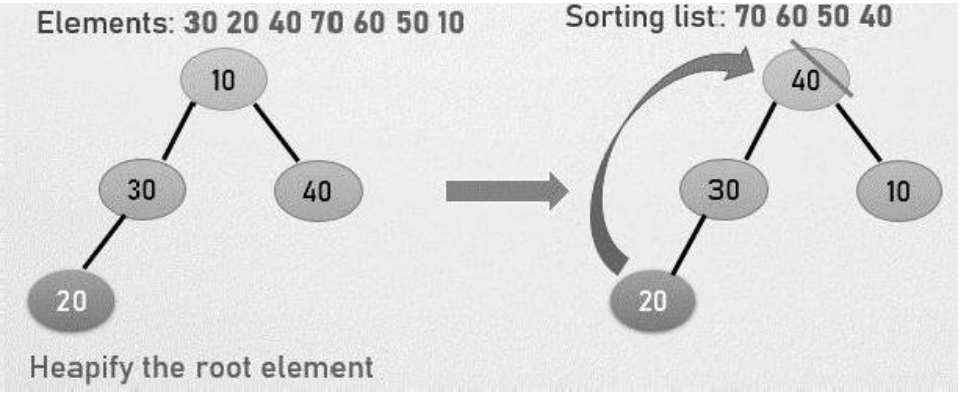
Step 2



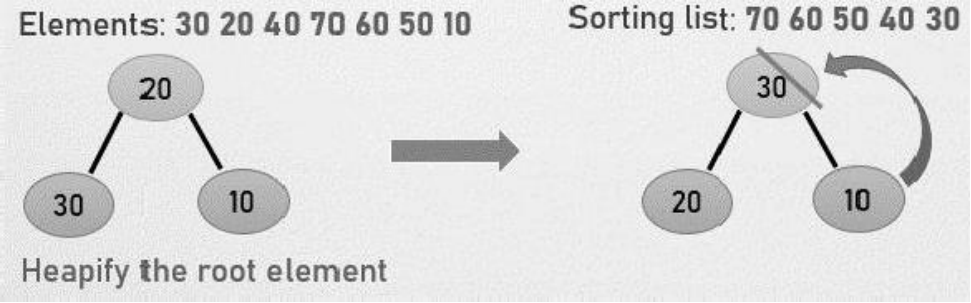
Step 3



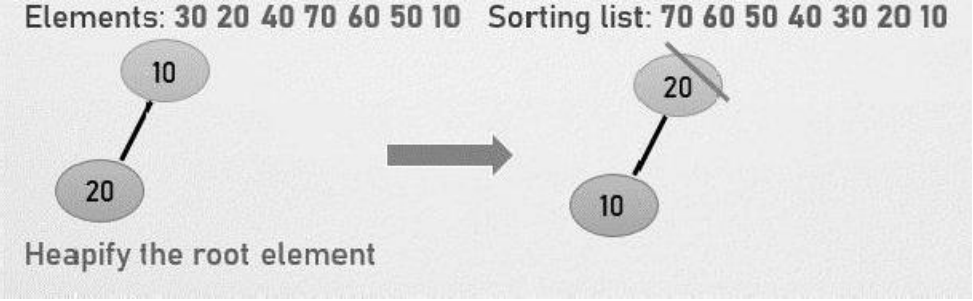
Step 4



Step 5



Step 6



Complexity of the Heap Sort

Worst Case: O(nlogn)

Best Case: O(n log n)

Average Case:O(n log n)

Applications

Systems and Embedded Systems: Heap sort is often used in systems where memory constraints are tight. Its in-place sorting characteristic, which requires only a constant amount of additional storage space, makes it ideal for embedded systems and real-time applications

Operating Systems: Heap sort is utilized in operating systems for job scheduling. The algorithm helps in efficiently managing the order of jobs by sorting them based on priority.

Network Traffic Management: In network systems, heap sort is used to manage packet scheduling. Sorting packets based on priority or arrival time ensures efficient handling of network traffic.

Event Simulation: In discrete event simulation, heap sort is used to manage the event queue. The algorithm helps in efficiently sorting and processing events in chronological order.

Data Stream Processing: Heap sort is useful in scenarios where a continuous stream of data needs to be sorted in real-time. The algorithm's efficiency in handling large data streams makes it suitable for such applications.

Priority Queue Implementation: Heap sort is foundational in implementing priority queues. Priority queues are widely used in various applications such as task scheduling, shortest path algorithms (like Dijkstra's), and bandwidth management.

Selection Algorithms: Heap sort is used in selection algorithms to find the k-th smallest (or largest) element in an array. The algorithm's ability to efficiently sort and maintain order is leveraged in these scenarios.

Graph Algorithms: Many graph algorithms, such as Prim's and Dijkstra's for finding minimum spanning trees and shortest paths, respectively, use heap sort or its underlying heap data structure for efficient edge and vertex selection.

External Sorting: Heap sort is useful in external sorting, where data that cannot fit into memory needs to be sorted. It is often used in combination with other algorithms to manage and merge large data sets.

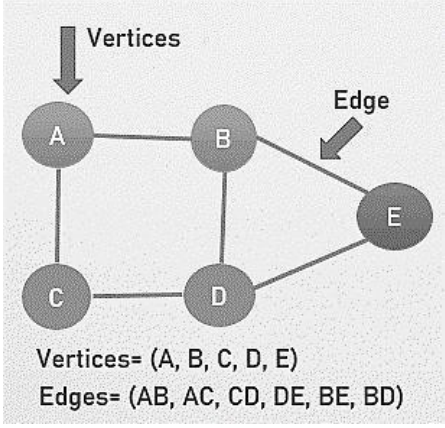
Database Management: In databases, heap sort is used to sort large datasets. Its efficiency in handling large volumes of data makes it suitable for indexing and query optimization.

Graphs

A Graph G consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and nonempty set of vertices. E is a set of pairs of vertices; these pairs are called edges. Therefore V(G), read as V of G, is set of vertices, and E(G), read as E of G, is set of edges.

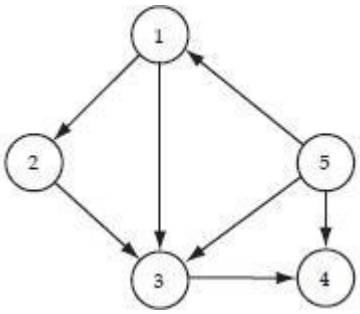
An edge e = (v,w), is a pair of vertices v and w, and is said to be incident with v and w. It is a pictorial representation of a set of objects where objects are connected by links. A graph may be pictorially represented as given in Figure

In an undirected graph, pair of vertices representing any edge is unordered. Thus (v,w) and (w,v) represent the same edge. In a directed graph each edge is an ordered pair of vertices, i.e. each edge is represented by a directed pair. If e = (v,w), then v is tail or initial vertex and w is head or final vertex. Subsequently (v,w) and (w,v) represent two



different edges.

A directed graph may be pictorically represented as given in Figure.



The direction is indicated by an arrow. The set of vertices for this graph remains the same as that of the graph in the earlier example,

i.e.V(G) = {(1,2,3,4,5)}

However the set of edges would be

E(G) = {(1,2), (2,3), (3,4), (5,4), (5,1), (1,3), (5,3)}

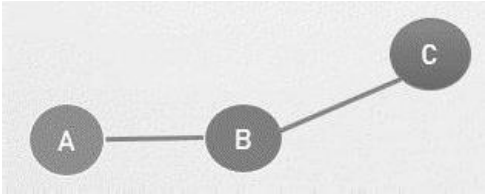
Graph Terminology

A good deal of nomenclature is associated with graphs. Most of the terms have straight forward definitions, and it is convenient to put them in one place even though we would not be using some of them until later.

Vertices: Each node of the graph is represented as a vertex.

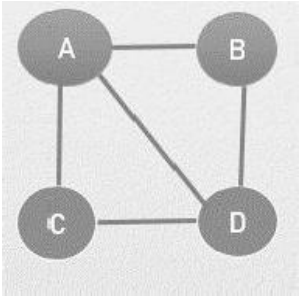
Edge: it is used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Path: Path represents a sequence of edges between the two vertices. E.g. ABC



Closed Path: A path will be called as closed path if the initial node is same as terminal node.

Degree of the Node: A degree of a node is the number of edges that are connected with that node. Degree of A=3.

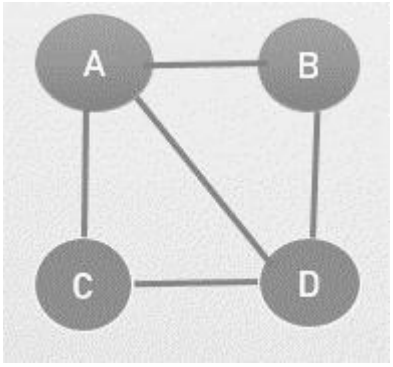


Adjacent Nodes/ Adjacency: if two nodes are connected to each other through an edge are called as neighbors or adjacent nodes.

Types of Graphs

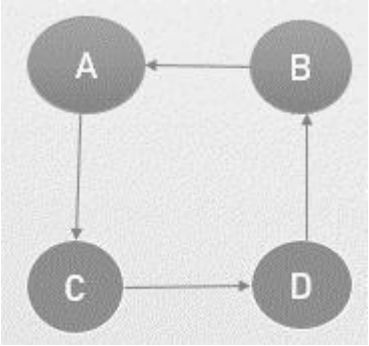
Undirected graph

An undirected graph nodes are connected and all the edges are bi-directional i.e. the edges do not point in any specific direction.



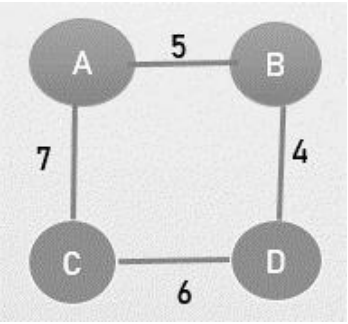
Directed graph

A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction. It is also called a digraph.



Weighted grahp

In weighted graph edges or path have values or cost. All the values seen associated with the edges are called weights.

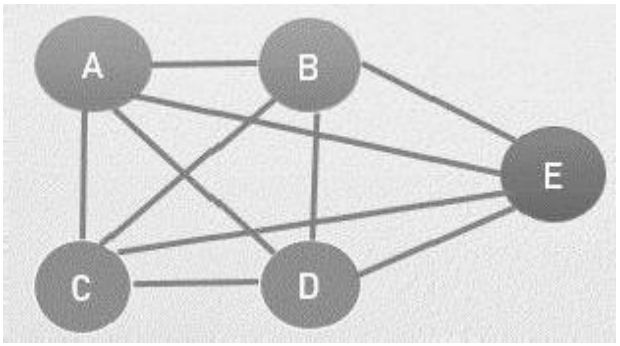


Un-weighted graph

In un-weighted graph there is no value or weight associated with the edge. By default, all the graphs are un-weighted.

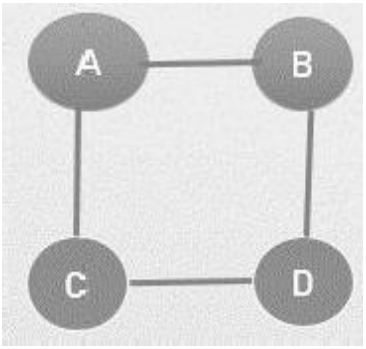
Complete graph

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



Finite graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number

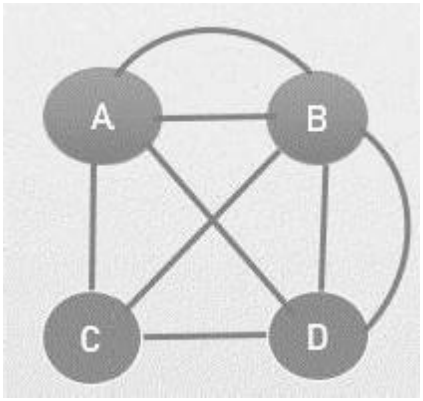


Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.

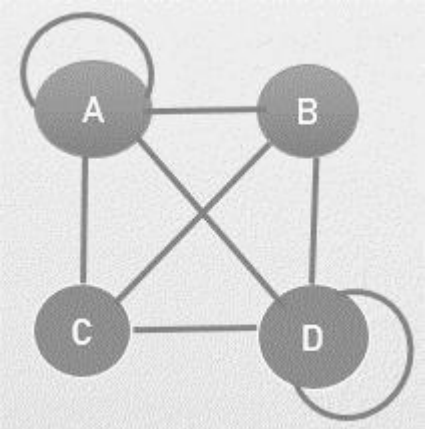
Multi Graph

If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multi graph. There are no self-loops in a Multi graph.



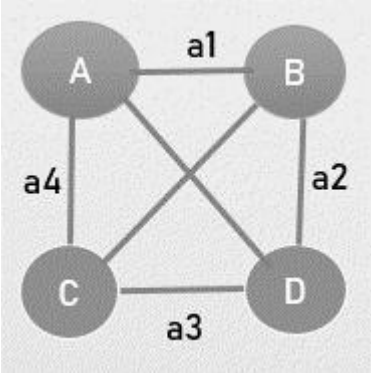
Pseudo graph

If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudo graph.



Labeled graph

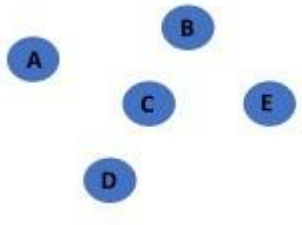
A graph G=(V, E) is called a labeled graph if its edges are labeled with some name or data.



Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them,

a graph G= (V, E) is a null graph



Representations of Graphs

Graph is a mathematical structure and finds its application in many areas of interest in which problems need to be solved using computers. Thus, this mathematical structure

must be represented as some kind of data structures. Two such representations are, commonly used.

There are various ways to represent a graph. A simple representation is given by an adjacency list, which specifies all vertices adjacent to each vertex of the graph. This list can be implemented as a table, in which case it is called a star representation, which can be forward or reverse.

Another representation is a matrix, which comes in two forms: an adjacency matrix and an incidence matrix. An adjacency matrix of graph G = (V,E) is a binary |V| × |V| matrix such that each entry of this matrix.

These are:

Adjacent Matrix

Adjacency List representation.

The choice of representation depends on the application and function to be performed on the graph.

Adjacent Matrix

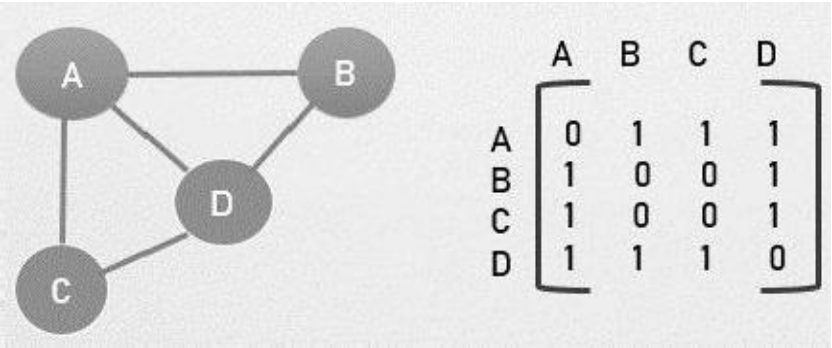
Two vertices is called adjacent or neighbor if it support at least one common edge. A finite graph can be represented in the form of a square matrix. Boolean value (0,1) of the matrix indicates if there is a direct path between two vertices.

It is also called 2D matrix that is used to map the association between the graph nodes. If a graph has n number of vertices, then the adjacency matrix of that graph is n x n, and each entry of the matrix represents the number of edges from one vertex to another.

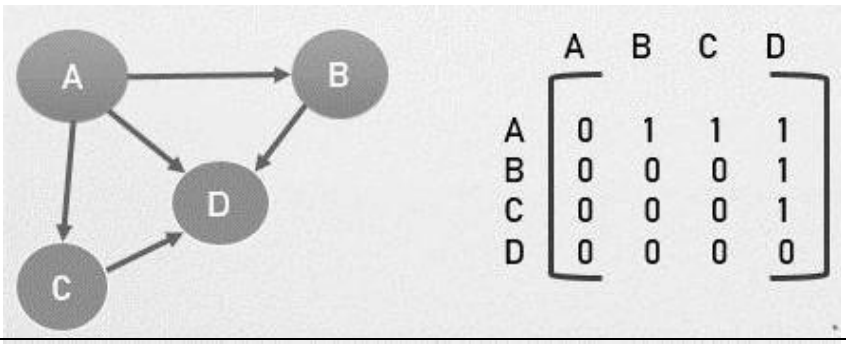
Adjacency Matrix Representation

The adjacency matrix A for a graph G = (V,E) with n vertices, is an n × n matrix of bits, such that A Aij = 1, iff there is an edge from vi to vj and Aij = 0, if there is no such edge.

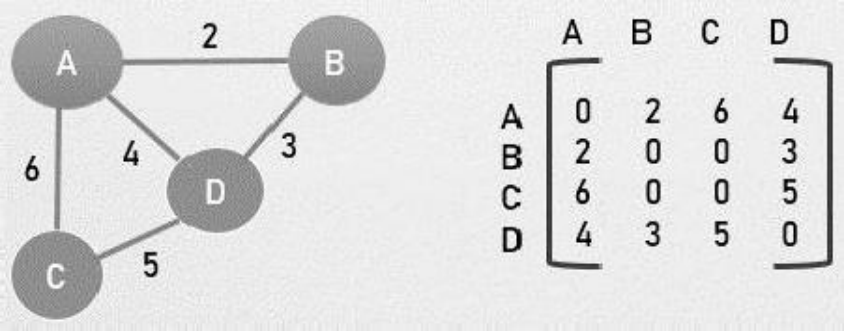
Undirected Graph Representation



Directed Graph Representation



Undirected Weighted Graph



Applications: Adjacency Matrix

Navigation tasks.

It is used to represent finite graphs.

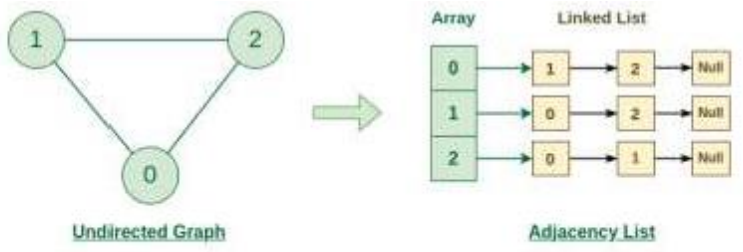
Creating routing table in networks.

Adjacency List Representation

A linked representation is an adjacency list.

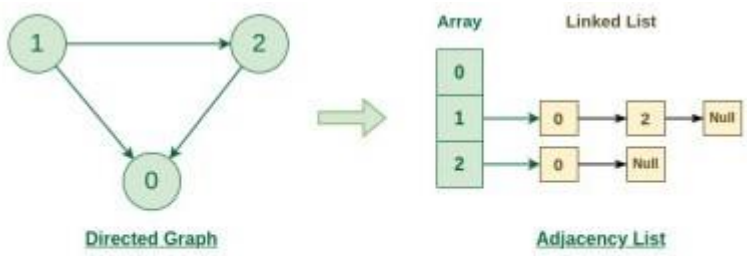
You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.

Representation of Undirected Graph to Adjacency list: The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex0 has two neighbours (i.e, 1 and2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex1,it has two neighbour (i.e,2and0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

Representation of Directed Graph to Adjacency list: The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Directed graph to Adjacency List

GRAPH TRAVERSAL

A graph traversal is a systematic way of visiting the nodes in a specific order. There are two types of graph traversal namely,

Depth first traversal

Breadth first traversal

Breadth First Traversal

Breadth First Search (BFS) of a graph G starts from an unvisited vertex u. Then all

The traversal terminates when there are no more nodes to visit. Breadth first search uses a queue data structure to keep track of the order of nodes whose adjacent nodes are to be visited.

Steps to implement breadth first search

Step 1: Choose any node in the graph, designate it as the search node and mark it

Step 2: Using the adjacency matrix of the graph, find all the unvisited adjacent nodes to the search node and enqueue them into the queue Q.

Step 3: Then the node is dequeued from the queue. Mark that node as visited and designate it as the new search node.

Step 4: Repeat step 2 and 3 using the new search node.

Step 5: This process continues until the queue Q which keeps track of the adjacent nodes is empty.

breadth first search algorithm

```
Algorithm BFS(v)
{
u:=v;
add u to q;
visited[v]:=1;
repeat
{
for all vertices w adjacent from u do
if(visited[w]=0) then
```

```
{
add w to q;
visited[w]=1;
}
}
if q is empty then turn;
delete u from q;
}
until(false);
}
```

DEPTH FIRST SEARCH

Depth first works by selecting one vertex V of G as a start vertex ; V is marked visited. Then each unvisited vertex adjacent to V is searched in turn using depth first search recursively. This process continues until a dead end (i.e) a vertex with no adjacent unvisited vertices is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.

The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth first search must be restarted at any one of them.

To implement the Depth first Search perform the following Steps :

Step : 1 Choose any node in the graph. Designate it as the search node and mark it as visited.

Step : 2 Using the adjacency matrix of the graph, find a node adjacent to the search. Node that has not been visited yet. Designate this as the new search node and mark it as visited.

Step : 3 Repeat step 2 using the new search node. If no nodes satisfying (2) can be found, return to the previous search node and continue from there.

Step : 4 When a return to the previous search node in (3) is impossible, the search from the originally choosen search node is complete

Step : 5 If the graph still contains unvisited nodes, choose any node that has not been visited and repeat step (1) through (4).

Depth First Search Algorithm

```
Algorithm dfs(v)
{
visited[v]=1;
for each vertex w adjacent from v do
{
If(visited[w]=0) then
dfs(w);
}
}
```

Applications of Depth First Search

To check whether the undirected graph is connected or not.

To check whether the connected undirected graph is Bi-connected or not.

To check the Acyclicity of the directed graph.

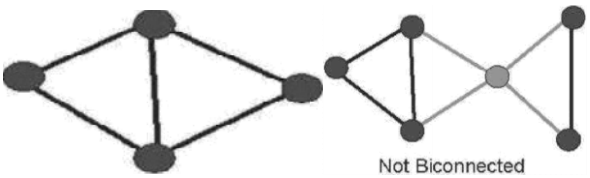
Bi-connected components:

A graph is said to be Bi-connected if:

It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.

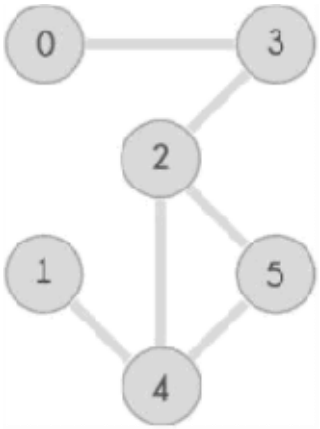
Even after removing any vertex the graph remains connected.

For example, consider the graph in the following figure



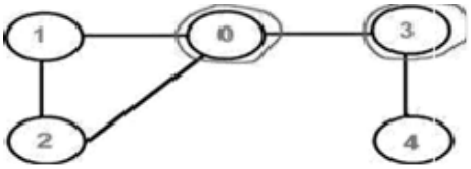
Bi-connected

Removing any of the vertices does not increase the number of connected components. So the given graph is Bi-connected.



In the above graph if the vertex 2 is removed, then here's how it will look:

Articulation point or cut point if a point in a graph becomes separated from the entire graph upon removal, it is referred to as an Articulation Point or Cut-Vertex.



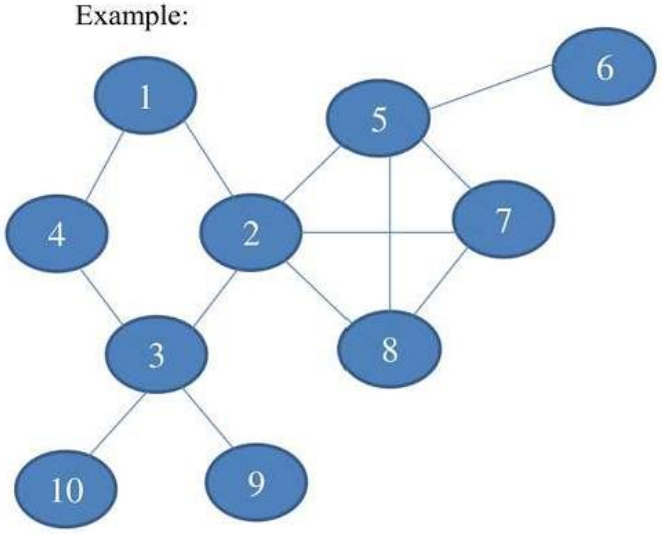
Construction of Bi-connected graph

Check the given graph whether it is bi-connected or not.

If the given graph is not bi-connected then identify all the articulation points.

If articulation points exist, determine a set of edges, whose inclusion makes the graph bi-connected.

Example:



The articulation points are 2, 3, and 5

To transform the graph into bi-connected graph, the new edges are included corresponding to the articulation point.

Edges corresponding to the articulation point 3- (4,l0)(10,9)

Edges corresponding to the articulation point 2-(1,5)(3,8)

Edges corresponding to the articulation point 5-(6,7)

2). Depth first spanning tree of the figure:-

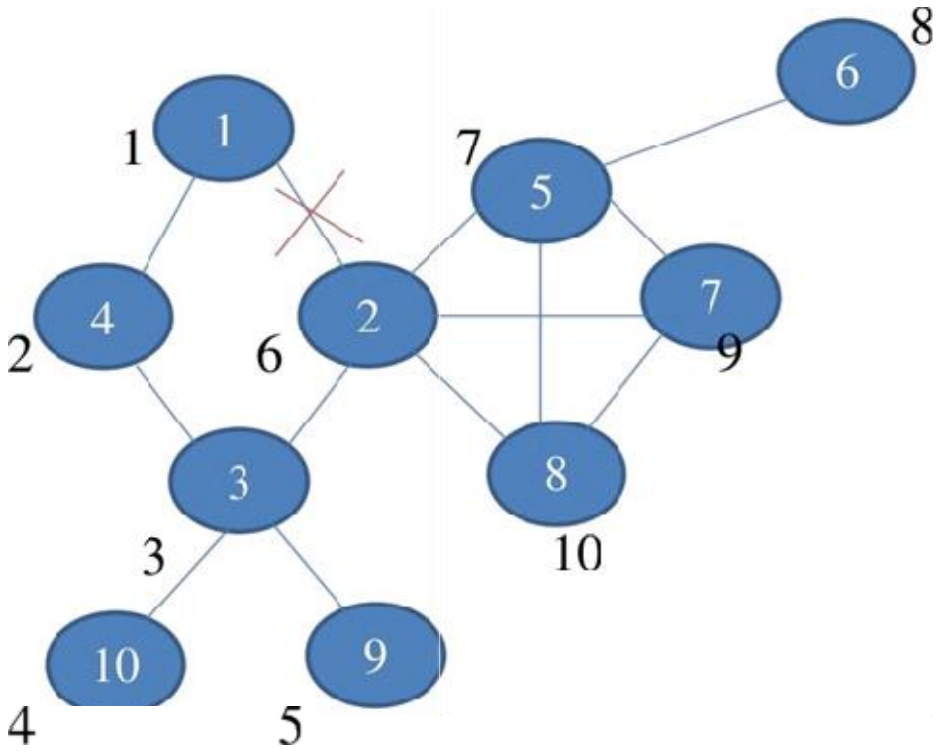


Fig:-Depth first spanning tree of above graph

In the fig there is a number of outside each vertex, corresponds to the order in which a DFS visits these vertices and are named as depth first numbers (dfn) of the vertex i. e

dfn[1]=1 dfn[2]=6

dfn[4]=2 dfn[5]=7

dfn[3]=3 dfn[6]=8

dfn[10]=4 dfn[7]=9

dfn[9]=5 dfn[8]=10

The solid edges of fig will form a depth first spanning tree. The depth first spanning tree representation is

Solid lines: Tree edges

-Dotted lines: Back edges

-Depth first spanning trees are very useful in identifying articulation points and bi-connected components

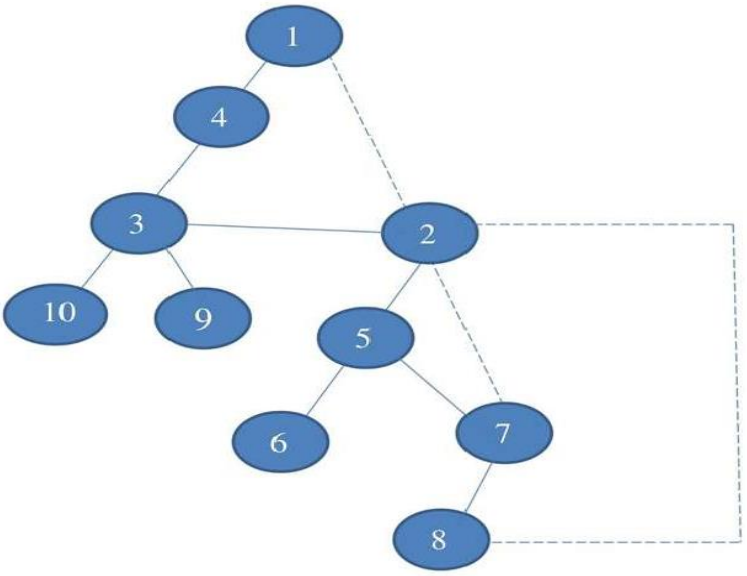


Fig:-Depth First Spanning Tree Reprsentation.

Depth First Spanning tree properties:-

If (u,v) is any tree edge of depth first spanning tree G, then either u is an ancestor of v or v is an ancestor of u.

Example:-

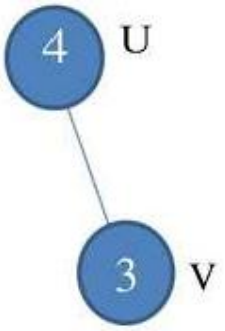
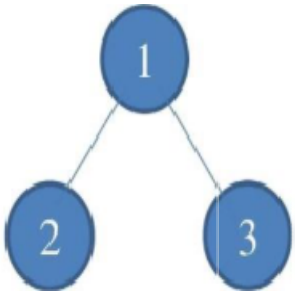


Fig:- A Tree With (U,V) Edge.

The root node of a depth first spanning tree is an articulation point if it has at least two children



Example:- Fig:- A Tree with Two Children.

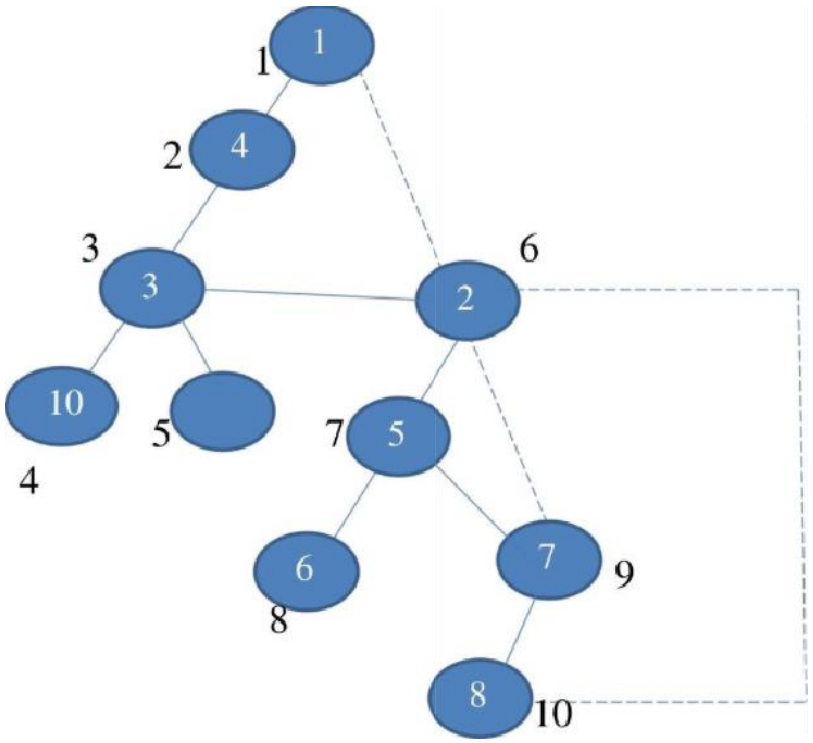
For each vertex, u, define L[U] as follows.

L[U] = min{dfn[U], min{L[W]/W is a child of U}, min{dfn[W]/(U,W) is a back edge } }

Where L[U] is the lowest depth first number that can be reached rom u using a path of descendants followed by at most one back edge.

If U is an articulation point iff U has a child W such that L[W] dfn[U]

Example:



L [10] = min {4,-,-}
L [10] = 4
L [09] = min {5,-,-}
L [09] = 5
L [06] = min {8,-,-}
L [06] = 8

L[08]=min {10,-,6}
L [08] = 6
L[07]=min {9,6,6}
L [07] = 6
L[05]=min {7,6,-}
L [05] = 6

$$L[02] = \min\{6,6,1\}$$
$$L[01] = 1$$
$$L[03] = \min\{3,1,-\}$$
$$L[03] = 1$$
$$L[04] = \min\{2,1,-\}$$
$$L[04] = 1$$
$$L[01] = \min\{1,1,-\}$$
$$L[02] = 1$$

(ie) L[1:10] = {1,1,1,1,6,8,6,6,5,4}

For the spanning tree of ig:- the articulation points are identified using the condition L(W) > dfn[U], iff U has a child W and U is not the root.

(1) For vertex (2) L[5] ≥ dfn [2]
6>6 = True = cut point

(2) For vertex (3) L[10] ≥ dfn [3]

(3) 4>3 = True = cut point

(4) For vertex (4) L[3] ≥ dfn [4]
1>2 = False

(5) For vertex (5) L[6] ≥ dfn [5]
8>7 = True = cut point

For vertex (7) L[8] ≥ dfn [7]
6>9 = False

The condition to check articulation point is not done for the vertices 6, 8, 9 and 10. Since w does not exist in the spanning tree structure and the condition is tme for the vertices 2, 3 and 5. Therefore the articulation points are 2, 3 and 5.

APPLICATIONS OF GRAPHS

1. **Social network graphs** to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

2. **Transportation networks** In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the

public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. **Utility graphs** The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

4. **Document link graphs** The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. **Protein-protein interactions graphs** Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

6. **Network packet traffic graphs** Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

7. **Scene graphs**. In graphics and computer games scene graphs represent the logical or special relationships between objects in a scene. Such graphs are very important in the computer games industry.

8. **Finite element meshes**: In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements form a graph that is called a finite element mesh.

9. **Robot planning**: Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous

10. **Neural networks**: Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when the human brain has about 1011 neurons.

Divide and conquer:

Introduction

General method: – Divide the problem instance into two or more smaller instances of the same problem, – Solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. – The recursion stops when an instance is reached which is too small to divide.

In Divide and Conquer approach, there are mainly three steps:

Step 1(Divide): The original problem is divided into one or more sub-problems in such a way that each sub-problem is equivalent to the original problem but its size is smaller than the original one. Further sub-division of each sub-problem is done till either it is directly solvable or it is impossible to perform sub-division which indicates that there is a direct solution of the sub-problem.

Step 2(Conquer): In this step, each smallest sub-problem is solved independently.

Step 3(Combine): The solution of each sub-problem is combined recursively to generate the solution of the original problem.

Algorithm

• Algorithm D and C(P)

• {

• if Small(P) then return S(P);

• else

• {

• divide P into smaller instances P1,P2,P3---- Pk, k≥1;

• Apply D and C to each of these sub problems;

• Return combine(D and C(P1),D and C(P2)---- ,D and C(Pk));

• }

• }

• Small(P) is a Boolean function that determines whether the input size is small enough that the answer can be computed without splitting.

Recurrence Relation

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n1) + T(n2) + \cdots T(nk) + f(n) & \text{otherwise} \end{cases}$$

The function g(n) is the time to compute the answer directly for small inputs.

• The function f(n) is the time for dividing the P and combining the solutions to sub-problems.

• The complexity of many divide-and-conquer algorithms is given by the recurrences of the form.

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

a and b are the known constants. T(1) is known and n is the power of b.

Solving the recurrence relation

substituting a=2, b=2 and T(1)=2

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

= 2[2T(n/4)+n/2]+n

=4T(n/4)+2n

=4[2T(n/8)+n/4]+2n

=8T(n/8)+3n . . .

=2i T(n/2 i)+i.n

Substitute n=2i => i=logn

=n T(1)+logn . n

=2n+nlogn=O(nlogn)

Applications of Divide and Conquer

Quick Sort

Merge Sort

Strassen's Matrix Multiplication

Convex Hull

Quick Sort

Quick Sort is a divide and conquer algorithm. This selects a pivot element and divides the array into two sub arrays.

Algorithm: Partition (a,m,p)

```
{
v=a[m];  i=m; j=p;
repeat
{
repeat
i=i+1;
until(a[i]≥v);
repeat
j=j-1;
until(a[j]≤v);
If(i<j)then interchange(a,i,j);
} until(i≥j);
a[m]=a[j]; a[j]=v; return j;
}
Algorithm interchange(a,i.j)
{
p=a[i];
a[i]=a[j];
a[j]=p;
}
Algorithm quicksort(p,q)
{
if(p<q) then
{
j=position(a,p,q+1)
```

```
quicksort(p,j-1);
quicksort (j+1,q);
}
}
```

Step by Step Process

In Quick sort algorithm, partitioning of the list is performed using following steps...

- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first positi ɔn in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] <= pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i >= j.
- **Step 7 -** Exchange the pivot elemnent with list[j] element.

Example

Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

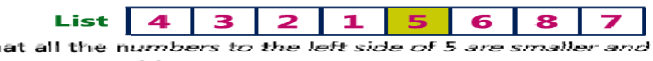
Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].

Compare List[left]<List[pivot] as it is true *increment left by one and repeat the same*, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

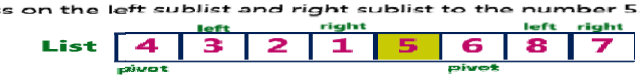
Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]



Here we can observe that all the *numbers to the left side of 5 are smaller and right* side are greater. That means 5 is placed in its correct position.
Repeat the same process on the *left sublist and right sublist* to the number 5.



In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).

Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...

Worst Case : O(n2)

Best Case : O (n log n)

Average Case : O (n log n)

Merge-Sort

Merge-Sort algorithm is a divide-and-conquer based sorting algorithm. It follows a divide and conquers approach to perform sorting. The algorithm in divide step repeatedly partitions an array into several sub arrays until each sub array consists of a single element. Then, it merges sub-problem solutions together according to following steps:

Divide the array into two equal sub-arrays Compare the sub-array's first elements

Conquer (solve) each sub-array by performing sorting operation. Apply the recursion unless the array is sufficiently small.

Combine (Merge) the solutions into a single array The following example illustrates the above steps:

Suppose the array has following numbers: 37 20 23 30 18 15 27 17

In divide step, the array is divided into two sub arrays 37 20 23 30 and 18 15 27 17

In conquer step, we sort each sub array 20 23 30 37 and 15 17 18 27

In combine (merge) all the sub arrays are merged in sorted order 15 17 18 20 23 27 30 37

Algorithm Merge sort (low, high)

{

If(low<high)then

{

Mid :=[(low+high)/2];

Merge sort (low, mid);

Merge sort (mid+1, high);

Merge (low, mid, high);

}}

Algorithm Merge (low, mid, high)

{

h:=low; i:=high; j:=mid+1;

While ((h≤ mid) and (j≤ high) h)

{

if(a[h] ≤a[j])then

{

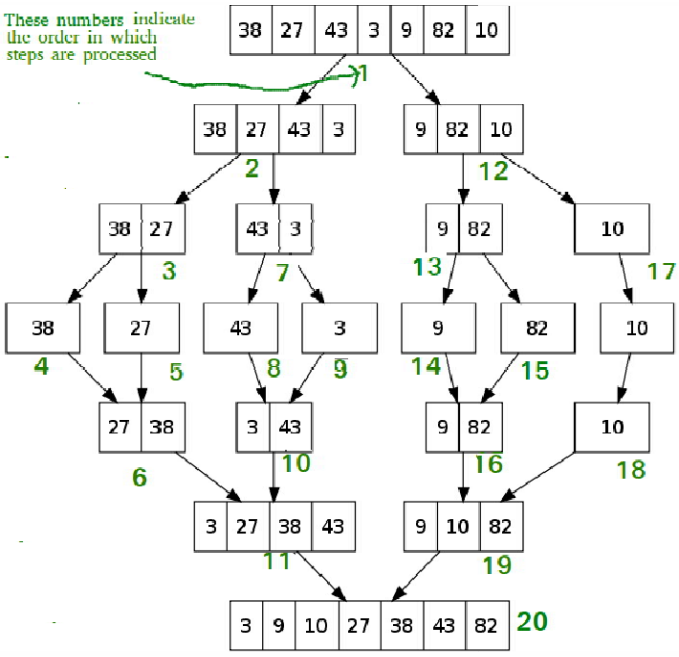
b[i]:=a[h];

h:=h+1;

}

```
else  {
b[i]:=a[j];
j:=j+1;  }
i:=i+1;
}
If (h>mid)then {
for k=j to high do {
b[i]:=a[k];
i:=i+1;
}  }
else {
for k=h to mid do {
b[i]:=a[k];
i:=i+1;
}
}
for  k:=low to high do
a[k]:=b[k]:
}
```

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Time Complexity:

The list of size N is divided into a max of logN parts, and the merging of all sublists into asingle list takes O(N) time, the worst case run time of this algorithm is O(NLogN)

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, j^{th} element is formed by taking elements in the i^{th} row of A and j^{th} column of B and multiplying them to get

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

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

C11=A11B11+A12B21

C12=A11B12+A12B22

C21=A21B11+A22B21

C22= A21B12+A22B22

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 Ci,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

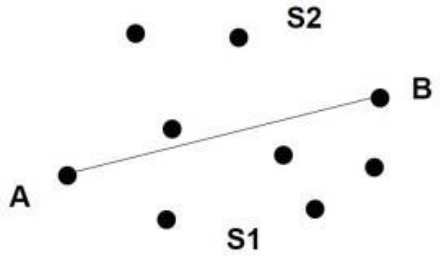
C22=P+R-Q+U

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

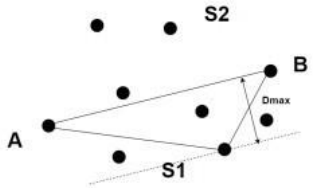
Convex Hull

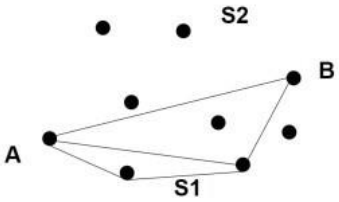
A convex hull is the smallest convex polygon that completely encloses a set of points in a two-dimensional or three-dimensional space. It can be thought of as the "envelope" or "wrapper" of the points. We use the divide and conquer approach to solve this by recursively calling the function for smaller parameters. It is a fundamental concept with applications in various fields such as computer graphics, robotics, and image processing. Convex Hull Problem deals with the problem of finding convex polygon with the minimum number of edges. Here is an illustration of our approach:

The first step is to find out the farthest two points in the plane:

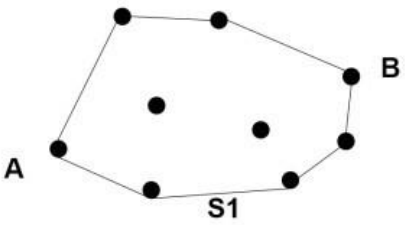


Then, in the two spaces S1 and S2, we will find out the farthest point:





Finally, Our resultant polygon would look something like this:



Importance of Convex Hull:

Convex hulls are important in computational geometry for several reasons:

Collision detection: Convex hulls can be used to efficiently detect collisions between objects in 2D or 3D space.

Image processing: Convex hulls can be used to extract meaningful shapes from images, such as the outline of an object.

Data visualization: Convex hulls can be used to visualize the distribution of data points in a scatter plot

Short Answer Questions

1. Define a Heap. Distinguish between Min Heap and Max Heap.
2. What is a Priority Queue? Mention any two applications.
3. State the Heap Property.
4. Define Heapify. What is its time complexity?
5. List the applications of Heap Sort.
6. Define Graph and its components.
7. Differentiate between Directed Graph and Undirected Graph.
8. Define Degree of a Vertex and Path in a Graph.
9. What is an Adjacency Matrix?
10. What is an Adjacency List? Mention one advantage.
11. Define Breadth First Search (BFS).
12. Define Depth First Search (DFS).
13. What is a Connected Component?
14. Define Bi-connected Graph.
15. What is an Articulation Point?
16. State the Divide and Conquer strategy.
17. What is the role of Pivot in Quick Sort?
18. State the best and worst-case time complexities of Quick Sort.
19. State the time complexity of Merge Sort.
20. Mention any two applications of Convex Hull.

Long Answer Questions

1. Explain the construction of a Max Heap with a suitable example. Discuss the insertion and deletion operations along with their time complexities.
2. **Q2.** Explain the Heap Sort algorithm with a suitable example. Derive its best, average, and worst-case time complexities.
3. **Q3.** Compare Min Heap and Max Heap. Explain the applications of Heaps and Priority Queues in real-world systems.
4. Explain Graph Terminology with suitable examples. Differentiate Directed and Undirected Graphs.
5. Explain Adjacency Matrix and Adjacency List representations of a graph. Compare them based on memory usage and traversal efficiency.
6. Explain the Breadth First Search (BFS) algorithm with an example. Analyze its time complexity.
7. Explain the Depth First Search (DFS) algorithm with an example. Discuss its applications.
8. Explain the concept of Connected Components and Bi-connected Components. Illustrate the role of articulation points using a suitable example.
9. Explain the Divide and Conquer strategy with a general algorithm and recurrence relation. Discuss its applications.
10. Explain the Quick Sort algorithm with an example. Analyze its best, average, and worst-case complexities.

Case Study – Unit II

Smart Emergency Response and Traffic Management System Using Heaps, Graphs, and Divide & Conquer Algorithms

Course

Algorithms, Data Structures and Algorithm Analysis (ADS&AA)

Difficulty Level

Intermediate

Learning Outcomes

After completing this case study, students will be able to:

- Apply Heap Trees and Priority Queues for scheduling.
- Model transportation networks using Graphs.
- Perform BFS and DFS traversals.
- Analyze connected and bi-connected components.
- Apply Divide and Conquer algorithms for efficient processing.
- Compare Quick Sort and Merge Sort.
- Identify suitable algorithms for real-world applications.

Background

A Smart City has developed an **Emergency Response System** to coordinate ambulances, fire engines, and police vehicles.

The city consists of

- 250 intersections
- 450 roads
- 15 hospitals
- 12 fire stations
- 10 police stations

Every emergency request must be processed immediately.

The system should

- Assign the nearest emergency vehicle.
- Find reachable hospitals.
- Detect road failures.
- Sort emergency requests according to priority.
- Process thousands of requests every day.

Problem Statement

The municipal corporation wants to design an intelligent system that can

- Handle emergency requests based on priority.
- Represent the road network efficiently.
- Identify disconnected regions caused by road closures.

- Find all reachable locations.
- Sort large emergency datasets quickly.
- Improve response time during disasters.

Existing System

Currently,

- Emergency requests are handled manually.
- Vehicles are assigned on a first-come-first-served basis.
- Road maps are checked manually.
- Searching hospitals takes more time.
- Sorting incident reports is slow.

Problems include:

- Delay in ambulance dispatch
- Traffic congestion
- Poor utilization of emergency vehicles
- Increased response time

Proposed Solution

Part A – Priority Queue Using Heap

Emergency requests are assigned priorities.

Emergency	Priority
Cardiac Arrest	Highest
Fire Accident	High
Road Accident	High
Theft	Medium
Complaint	Low

A **Max Heap** is used as a Priority Queue to ensure that the highest-priority request is always processed first. Heap operations such as insertion, deletion, and extraction take **O(log n)** time, making them suitable for real-time scheduling.

Part B – Road Network Using Graphs

The city map is represented as a graph:

- Vertices → Road intersections
- Edges → Roads
- Edge weights → Distance or travel time

Graph representations using adjacency matrices or adjacency lists allow efficient storage and processing depending on the application.

Part C – Breadth First Search (BFS)

When an ambulance is dispatched, the system uses **BFS** to explore all nearby intersections level by level to identify the nearest hospital in an unweighted road network.

Advantages:

- Finds nearby locations
- Efficient exploration
- Useful in shortest-path searches for unweighted graphs

Part D – Depth First Search (DFS)

During disasters such as floods or earthquakes, DFS is used to determine whether every area remains reachable and to identify isolated regions. DFS also helps detect connected and bi-connected components.

Applications:

- Connectivity checking
- Route exploration
- Disaster recovery planning

Part E – Bi-connected Components

Some intersections act as **articulation points**. If such an intersection is blocked, parts of the city may become disconnected.

The city analyzes **bi-connected components** to identify critical intersections and add alternate roads, thereby improving network reliability.

Part F – Divide and Conquer

The emergency control center receives thousands of incident records every day.

To process this data efficiently:

Quick Sort

Quick Sort rapidly partitions and sorts emergency records based on severity or time. It has an average complexity of **O(n log n)**, though the worst case is **$O(n^2)$**.

Merge Sort

Merge Sort divides records into smaller groups, sorts them, and merges the results. Its time complexity remains **O(n log n)** in all cases, making it reliable for large datasets.

Algorithm Analysis

Algorithm	Application	Complexity
Heap Insert/Delete	Emergency scheduling	O(log n)
BFS	Find nearest hospital	O(V + E)
DFS	Connectivity analysis	O(V + E)
Quick Sort	Sort emergency reports	Average O(n log n)
Merge Sort	Large-scale record sorting	O(n log n)

Additional Applications

The system can also incorporate:

- **Strassen's Matrix Multiplication** for fast matrix operations in traffic simulations and large network computations.
- **Convex Hull** to determine the boundary of disaster-affected areas for rescue planning and resource allocation.

Real-World Applications

These algorithms are widely used in:

- Google Maps
- GPS Navigation Systems
- Uber and Ola
- Food Delivery Platforms
- Airline Reservation Systems
- Smart Traffic Management
- Hospital Emergency Services
- Disaster Management Systems
- Network Routing
- Social Networks

Discussion Questions

1. Why is a Heap-based Priority Queue more suitable than a normal queue for emergency dispatch systems?
2. Explain how graphs represent a city's transportation network.
3. Compare BFS and DFS for emergency route planning.
4. Why are articulation points critical in transportation networks?
5. Compare Quick Sort and Merge Sort in terms of efficiency and stability.
6. Explain how Divide and Conquer improves the performance of sorting algorithms.
7. Suggest how Convex Hull can assist disaster management.
8. Why is maintaining bi-connected components important in smart city infrastructure?

Conclusion

The Smart Emergency Response System demonstrates how **Heap Trees**, **Priority Queues**, **Graphs**, **BFS**, **DFS**, **Bi-connected Components**, and **Divide and Conquer algorithms** work together to build a fast, reliable, and scalable emergency management solution. Using efficient data structures and algorithms reduces response time, improves traffic management, and ensures uninterrupted connectivity even during critical situations.

UNIT-III

Greedy Method : General Method

Greedy method is the most straightforward designed technique.

As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

Definition:

A problem with N inputs will have some constraints any subsets that satisfy these constraints are called a **feasible solution.**

A feasible solution that either maximize or minimize a given objectives function is called an **optimal solution.**

General Method:

Control abstraction algorithm for Greedy Method:

1. Algorithm Greedy (a,n)
2. //a[1:n] contain the 'n' inputs

```
3. {
4.Solution =0 ;  //Initialise the solution.
5. For i:=1 to n do
6. {
7. x:=select(a);
8. if(feasible(solution,x))then
9. solution=union(solution, x);
10.}
11. return solution;
12. }
```

The function select an input from a[] and removes it. The select input value is assigned to X. Feasible is a Boolean value function that determines whether X can be included into the solution vector.

The function Union combines X with The solution and updates the objective function. The function Greedy describes the essential way that a greedy algorithm will once a particular problem is chosen and the function subset, feasible & union are properly implemented.

2.1. Knapsack Problem

We are given **n** objects and knapsack or bag with capacity m object **i** has profit Pi and weight Wi where

i varies from **1 to n**.

The problem is we have to fill the bag with the help of **n** objects and the resulting profit has to

be maximum.

Formally the problem can be stated as

Maximize $\sum_{1<=i<=n} x_i p_i$

subject to $\sum_{1<=i<=n} x_i w_i <= m$

Where **xi** is the fraction of object and it lies between 0 to 1.

There are so many ways to solve this problem, which will give many feasible solution for which we have to find the optimal solution. But in this algorithm, it will generate only one

Solution which is going to be feasible as well as optimal. First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios.

Select an object with highest p/w ratio and check whether its height is lesser than

the capacity of the bag.

If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.

Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected .in this case place a fraction of the object and come out of the loop.Whenever you selected.

ALGORITHM:

Algorithm Greedy knapsack (m,n) 2//P[1:n] and

the w[1:n]contain the profit3.// Profits&

weight's.of the n object ordered.4.//such that

p[i]/w[i] >=p[i+1]/W[i+1]

5. //n is the Knapsack size and x[1:n] is the solution vertex.
6.{

7. for I=1 to n do a[I]=0.0; 8.U=n;

9. For I=1 to n do

10.{

11.if (w[i]>u)then break;

13.x[i]=1.0;U=U-w[i]

14.}

15.if(i<=n)then

x[i]=U/w[i];16.}

Example:

Capacity=20

N=3,M=20

Wi=18,15,10

Pi=25,24,15

Pi/Wi=25/18=1.38, 24/15=1.6, 15/10=1.5

Descending Order Pi/Wi1.6 1.5 1.38

Pi = 24 15 25

Wi = 15 10 18

Xi = 1 5/10 0

PiXi=1*24+0.5*15 31.5

The optimal solution is 31.5

X1	X2	X3	WiXi	PiXi
½	1/3	¼	16.6	24.25
1	2/5	0	20	18.2
0	2/3	1	20	31
0	1	½	20	31.5

Of these feasible solution Solution 4 yield the Max profit .As we shall soon see this solution is optimal for the given problem instance.

Job Sequencing with deadlines:

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

Points To remember:

To complete a job, one has to process the job or an action for one unit of time. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.

If we select a job at that time ,since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

So the waiting time and the processing time should be less than or equal to the dead line of the job.

ALGORITHM:

```
Algorithm JS(d,j,n)
//The job are ordered such that p[1]>p[2]…>p[n] //j[i] is the ith job in
        the optimal solution
//  Also at terminal d [ J[ i]<=d[ J {i+1],1<i<k
{
d[0]= J[0]=0;
{ // consider jobs in decreasing order of P[I];find the position for I and check feasibility insertion
r=k;
while((d[J[r]]>d[i] )and
(d[J[r]] = r)do r =r-1;
if (d[J[r]]<d[I])and (d[I]>r))then
{
for q=k to (r+1) step –1 do J [q+1]=j[q] J[r+1]=i;

}
}
return k;
}
```

Without feasibility rule:

Example1. n=5 (P1,P2,…P5)=(20,15,10,5,1)

(d1,d2….d3)=(2,2,1,3,3)

Sno	feasible solution	Processing Sequence	Value
1.	(1)	(1)	20
2.	(2)	(2)	15
3.	(3)	(3)	10
4.	(4)	(4)	5
5.	(5)	(5)	1
6.	(1,2)	(2,1)	35

7. (1,3)	(3,1)	30
8. (1,4)	(1,4)	25
9.(1,5)	(1,5)	21
10.(2,3)	(3,2)	25
11.(2,4)	(2,4)	20
12.(2,5)	(2,5)	16
13.(1,2,3,3	(3,2,1)	45
14. (1,2,4)	(1,2,4)	40

The Solution 13 is optimal

Example 4.2 Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

By Using feasibility rule:

Example 4.3 Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

2.3 .MINIMUM COST SPANNING TREE

Let G(V,E) be an undirected connected graph with vertices 'v' and edge 'E'.A sub-graph t=(V,E') of the

G is a Spanning tree of G iff 't' is a tree.

The problem is to generate a graph G'= (V,E) where 'E' is the subset of E,G' is a Minimum spanning tree.Each and every edge will contain the given non-negative length .connect all the nodes with edge present in set E' and weight has to be minimum.

NOTE:

We have to visit all the nodes.

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

Definition:

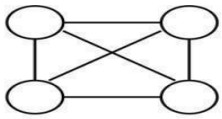
A spanning tree of a graph is an **undirected** tree consisting of only those edge that are necessary to connect all the vertices in the original graph.

A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

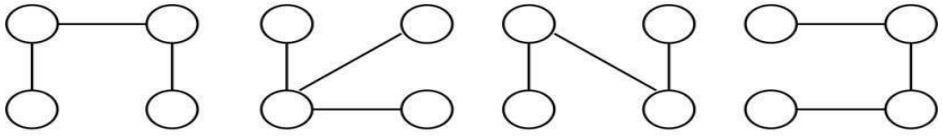
SPANNING TREE: - A Sub graph 'n' of o graph 'G' is called as a spanning tree if

(i) It includes all the vertices of 'G'

(ii) It is a tree



A connected, undirected graph



Four of the spanning trees of the graph

Application of the spanning tree:

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

2.3.1 METHODS OF MINIMUM COST SPANNING TREE:

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

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

2.3.2 Kruskal's algorithm:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

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

To find the minimum cost spanning tree the edge are inserted to tree in increasing-order of their cost

Algorithm:

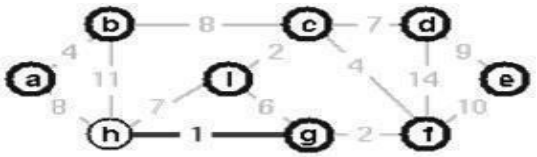
```
Algorithm kruskal(E,cost,n,t)
//E    set of edges in G has 'n' vertices.
//cost[u,v]      cost of edge (u,v).t        set of edge in minimum cost spanning tree
//  the first cost is returned.
{
for i=1 to n do parent[I]=-1; I=0;
mincost=0.0;
While((I<n-1)and (heapnotempty)) do
{
j=find(n);
k=find(v);
if(j not equal k) than
{
i=i+1
```

```
t[i,1]=u;

t[i,2]=v;
mincost=mincost+cost[u,v];

union(j,k);

      }
   }
```

if(i notequal n-1) then write("No spanning tree")

else return minimum cost;

}

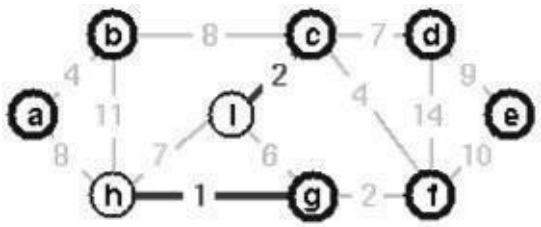
An  spanning tree algorithm in worst case is

where E is the edge set of G.

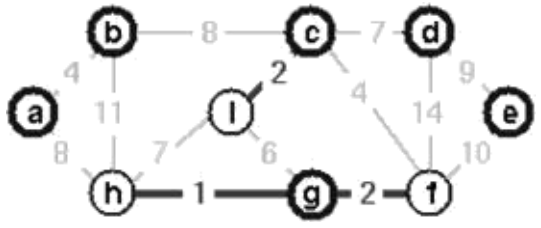
Example: Step by Step operation of Kurskal algorithm.

Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Let's choose vertex g arbitrarily.

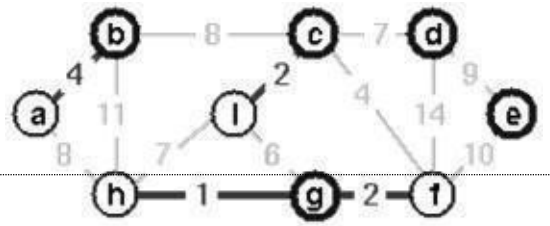
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



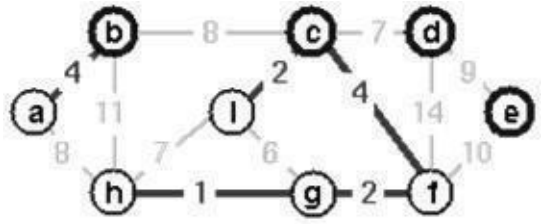
Step 3. Edge (g, f) is the next shortest edge. Add this edge and choose vertex g as representative.



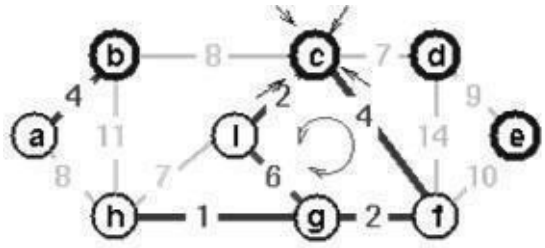
Step 4. Edge (a, b) creates a third tree.



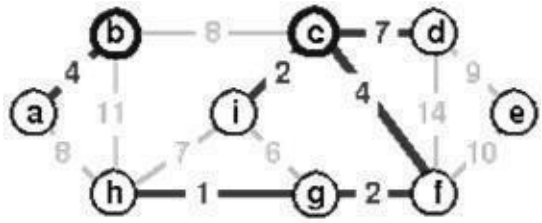
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



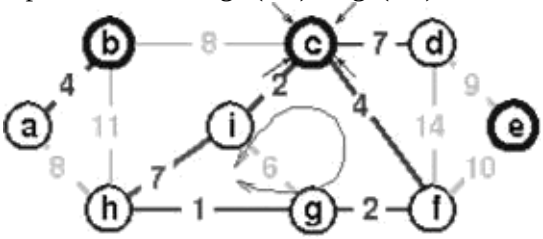
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



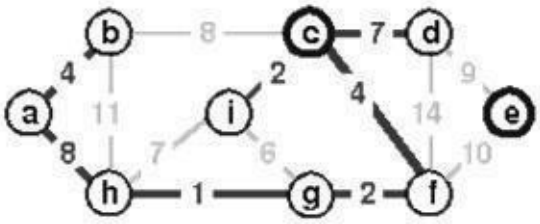
Step 7. Instead, add edge (c, d).



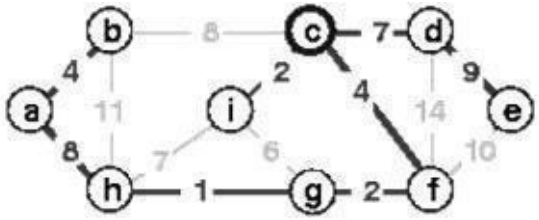
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



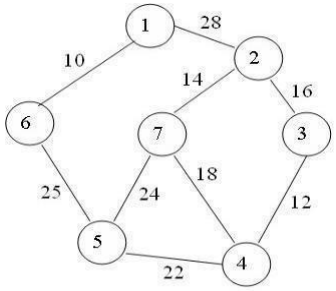
Step 9. Instead of adding edge (h, i) add edge (a, h).



Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



2.3.3 PRIM'S ALGORITHM



Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have **been** reached.

```
Algorithm prims(e,cost,n,t)
{
  Let (k,l) be an edge of minimum cost in E;
  Mincost :=cost[k,l];
  T[1,1]:=k; t[1,2]:=l;
  For I:=1 to n do
     If (cost[i,l]<cost[i,k]) then near[i]:=l;
     Else near[i]:=k;
 Near[k]:=near[l]:=0;
 For i:=2 to n-1 do
  {
    Let j be an index such that near[j]≠0 and
    Cost[j,near[j]] is minimum;
    T[i,1]:=j; t[i,2]:=near[j];
    Mincost:=mincost+ Cost[j,near[j]];
    Near[j]:=0;
    For k:=0 to n do
       If near((near[k]≠0) and (Cost[k,near[k]]>cost[k,j])) then
      Near[k]:=j;
  }
```

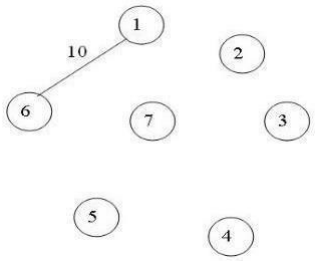
Return mincost;

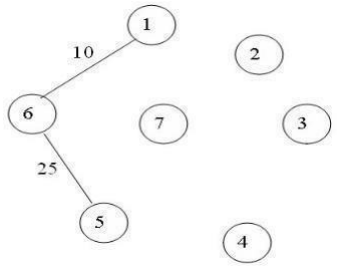
}

The prims algorithm will start with a tree that includes only a minimum cost edge of G. Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.

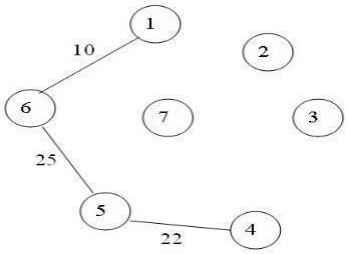
The working of prims will be explained by following diagram

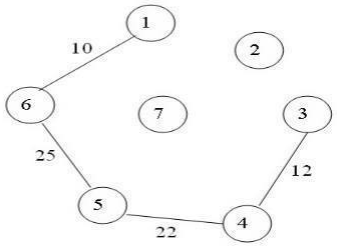
Step 1: Step 2:

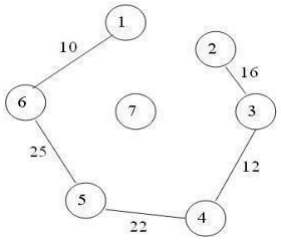


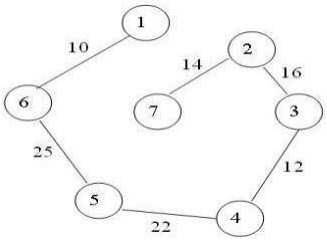


Step 3: **Step 4:**









Step 5: **Step 6**

2.4 SINGLE-SOURCE SHORTEST PATH:

Heres how Dijkstras algorithm ,a greedy approach works:

Initialization:

Assign a distance of 0 to the source vertex and inifinity to all other vertices.

Create a set of visited vertices ,initially empty

Iterarion:

- While there are unvisited vertices
- Select the unvisited vertex with the smallest current distance. This is the greedy choice – always pick the locally optimal (shortest distance) path so far

- Mark the selected vertex as visited.
- For each unvisited neighbor of the selected vertex:
- Calculate the distance from the source to the neighbor through the selected vertex (current distance of selected vertex + weight of edge to neighbor)

Termination:

The algorithm terminates when all vertices have been visited, or when the destination vertex (if a specific one is sought) has been visited. At this point, the recorded distances for all vertices represent the shortest paths from the source

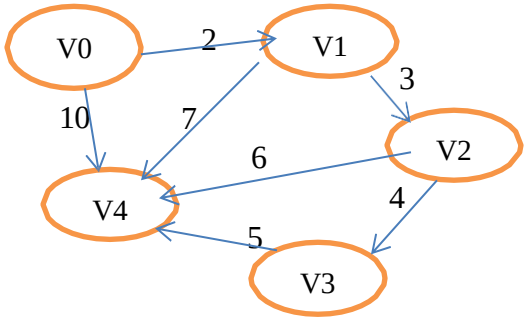
Why it is a Greedy Method:

Dijkstra's algorithm is considered greedy because at each step, it makes the locally optimal choice by selecting the unvisited vertex with the smallest known distance from the source. This local optimization ultimately leads to a globally optimal solution for graphs with non-negative edge weights.

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

- Is there a path from A to B?
- If there is more than one path from A to B? Which is the shortest path

Example1:



S	Solution set	Selected vertex	V0	V1	V2	V3	V4

1	-	V0	0	2	∞	∞	10
2	{ vo}	V1	0	2	5	∞	9
3	{ vo,v1}	V2	0	2	5	9	∞
4	{ vo,v1,v2}	V3	0	2	5	9	9
5	{ vo,v1,v2,v3}	V4	0	2	5	9	9
6	{ vo,v1,v2,v3,v4}	-	0	2	5	9	9

Finally the cheapest path from v0 to all other vertices is given by V1 V2 V3 V4

Second example:

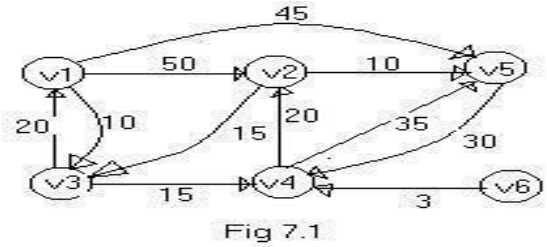


Fig 7.1

The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph

G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of fig 7- 1. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

- v1 v2 = 50 units
- v1 v3 v4 v2 = 10+15+20=45 units
- v1 v5 v4 v2 = 45+30+20= 95 units
- v1 v3 v4 v5 v4 v2 = 10+15+35+30+20=110 units

To formulate a greedy based algorithm to generate The cheapest path among these is the path along v1 v3 v4 v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1 v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.

the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.

```
1.Algorithm Shortest Paths(u, cost, dist, n)
2 // dist[j]1,<j <n, is set to the length of the shortest
3 // path from vertexv to vertexj in a digraphG with n
4 // vertices.dist[v] is set to zero.G is represented by its
5 // costadjacencymatrixcost[l :n, 1:n].
6 {
7 for i :=1to n do
8 {// InitializeS.
9 S[i]:=false;dist[i]:=cost[v,i];
10 }
11 S[v]:=true;dist[v] :=0.0;// Put v in 5.
12 for num :=2 to n-1do
13 {
14 // Determine n-1paths from v.
15 Choose u from among those vertices not
```

```
16 in Ssuch that dist[u]is minimum;
17 S[v]:=true;// Put u in S.
18 for (each w adjacent to u with S[w]= false)do
19 // Update distances.
20 if {dist[w]>dist[u]+cost[u,w])) then
21 dist[w] :=dist[u]+cost[u,w];
22 }
23 }
```

Algorithm: Greedy algorithm to generate shortest paths

Dynamic Programming: General Method, All pairs shortest paths, Optimal Binary Search Trees, 0/1Knapsack, String Editing, Travelling Salesperson problem.

DYNAMIC PROGRAMING GENERAL METHOD:

1. The idea of dynamic programming is thus quit simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.
2. Divide and conquer is a top-down method.
3. When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub- instances as the algorithm progresses.
4. Dynamic programming on the other hand is a bottom-up technique.
5. We usually start with the smallest and hence the simplest sub-instances.
6. By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances.
7. The essential difference between the greedy method and dynamic programming is that the greedy method only one decision sequence is ever generated.
8. In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

All pairs shortest paths:

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as cell as one from j to i. These two paths are the same when G is undirected.When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the ength of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n2) time, the matrix A can be obtained in O (n3) time.

The dynamic programming solution, called Floyd's algorithm, runs in O (n3) time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).The shortest i to j path in G, $i \neq j$ originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let A k (i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

A k (i, j) = {min {min {Ak-1 (i, k) + Ak-1 (k, j)}, c (i, j)}

1<k<n

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n3).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.

cost adjacency

matrix (A^0)=

$$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

General formula: min $\{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$
$1<k<n$
Solve the problem for different values of k = 1, 2 and 3

Step 1: Solving the equation for, k = 1;

$A^1(1, 1) = \min\{(A^o(1, 1) + A^o(1, 1)), c(1, 1)\} = \min\{0 + 0, 0\} = 0$
$A^1(1, 2) = \min\{(A^o(1, 1) + A^o(1, 2)), c(1, 2)\} = \min\{(0 + 4), 4\} = 4$
$A^1(1, 3) = \min\{(A^o(1, 1) + A^o(1, 3)), c(1, 3)\} = \min\{(0 + 11), 11\} = 11$
$A^1(2, 1) = \min\{(A^o(2, 1) + A^o(1, 1)), c(2, 1)\} = \min\{(6 + 0), 6\} = 6$
$A^1(2, 2) = \min\{(A^o(2, 1) + A^o(1, 2)), c(2, 2)\} = \min\{(6 + 4), 0)\} = 0$
$A^1(2, 3) = \min\{(A^o(2, 1) + A^o(1, 3)), c(2, 3)\} = \min\{(6 + 11), 2\} = 2$
$A^1(3, 1) = \min\{(A^o(3, 1) + A^o(1, 1)), c(3, 1)\} = \min\{(3 + 0), 3\} = 3$
$A^1(3, 2) = \min\{(A^o(3, 1) + A^o(1, 2)), c(3, 2)\} = \min\{(3 + 4), \quad\} = 7$
$A^1(3, 3) = \min\{(A^o(3, 1) + A^o(1, 3)), c(3, 3)\} = \min\{(3 + 11), 0\} = 0$

$A^{(1)} =$

$$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

Step 2: Solving the equation for, K = 2;

$A^2(1, 1) = \min\{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min\{(4 + 6), 0\} = 0$
$A^2(1, 2) = \min\{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min\{(4 + 0), 4\} = 4$
$A^2(1, 3) = \min\{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min\{(4 + 2), 11\} = 6$
$A^2(2, 1) = \min\{(A^1(2, 2) + A^1(2, 1), c(2, 1)\} = \min\{(0 + 6), 6\} = 6$
$A^2(2, 2) = \min\{(A^1(2, 2) + A^1(2, 2), c(2, 2)\} = \min\{(0 + 0), 0\} = 0$
$A^2(2, 3) = \min\{(A^1(2, 2) + A^1(2, 3), c(2, 3)\} = \min\{(0 + 2), 2\} = 2$
$A^2(3, 1) = \min\{(A^1(3, 2) + A^1(2, 1), c(3, 1)\} = \min\{(7 + 6), 3\} = 3$
$A^2(3, 2) = \min\{(A(3, 2) + A(2, 2), c(3, 2)\} = \min\{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min \{A^1(3, 2) + A^1(2, 3), c(3, 3)\} = \min \{(7 + 2), 0\} = 0$

$$A^{(2)} = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

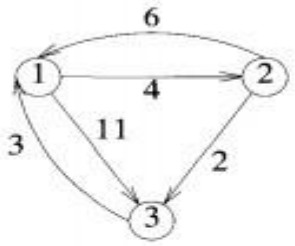
Step 3: Solving the equation for, k = 3;

$A^3(1, 1) = \min \{A_2(1, 3) + A_2(3, 1), c(1, 1)\} = \min \{(6 + 3), 0\} = 0$
$A^3(1, 2) = \min \{A_2(1, 3) + A_2(3, 2), c(1, 2)\} = \min \{(6 + 7), 4\} = 4$
$A^3(1, 3) = \min \{A_2(1, 3) + A_2(3, 3), c(1, 3)\} = \min \{(6 + 0), 6\} = 6$
$A^3(2, 1) = \min \{A_2(2, 3) + A_2(3, 1), c(2, 1)\} = \min \{(2 + 3), 6\} = 5$
$A^3(2, 2) = \min \{A_2(2, 3) + A_2(3, 2), c(2, 2)\} = \min \{(2 + 7), 0\} = 0$
$A^3(2, 3) = \min \{A_2(2, 3) + A_2(3, 3), c(2, 3)\} = \min \{(2 + 0), 2\} = 2$
$A^3(3, 1) = \min \{A_2(3, 3) + A_2(3, 1), c(3, 1)\} = \min \{(0 + 3), 3\} = 3$
$A^3(3, 2) = \min \{A_2(3, 3) + A_2(3, 2), c(3, 2)\} = \min \{(0 + 7), 7\} = 7$
$A^3(3, 3) = \min \{A_2(3, 3) + A_2(3, 3), c(3, 3)\} = \min \{(0 + 0), 0\} = 0$

$$A^{(3)} = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

This is resultant cost path matrix

Example 5.15 The graph of Figure 5.6(a) has the cost matrix of Figure 5.6(b). The initial A matrix, $A^{(0)}$, plus its values after 3 iterations $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ are given in Figure 5.6. □



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Figure 5.6 Directed graph and associated matrices

Optimal Binary Search Trees:

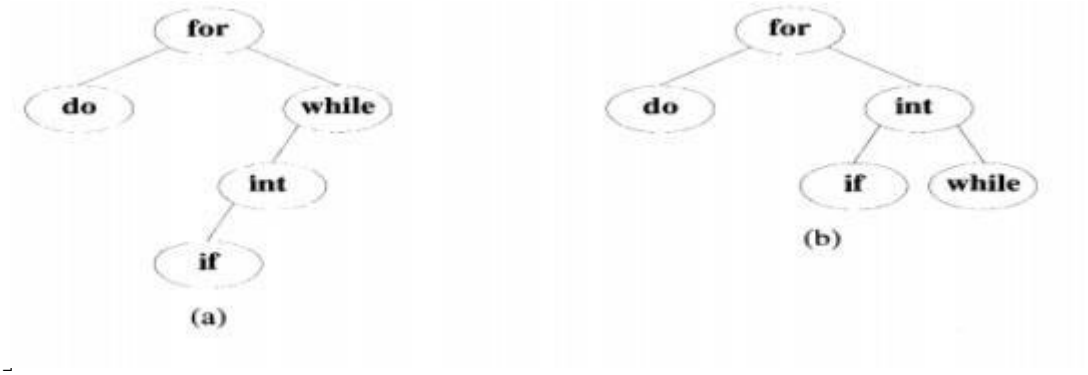


Fig:Two Possible Binary search trees

Suppose we are searching a word from a dictionary, and for every required word, we are looking up in the dictionary then it becomes time consuming process.the binary search tree of common words as key elements. Again we can make this binary search tree efficiently by arranging frequently words nearer to the root and less frequently words away from the root. Such a binary search tree makes out task more simplified as well as efficient. This type of binary search tree is called optimal binary search tree(OBST).

Example 5.18 Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ **(do, if, int, while)**. Let $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i,i) = q(i), c(i,i) = 0$ and $r(i,i) = 0, 0 \le i \le 4$. Using Equation 5.12 and the observation $w(i,j) = p(j) + q(j) + w(i,j-1)$, we get

$$
\begin{array}{rcl}
w(0,1) & = & p(1) + q(1) + w(0,0) = 8 \\
c(0,1) & = & w(0,1) + \min\{c(0,0) + c(1,1)\} \quad = \quad 8 \\
r(0,1) & = & 1 \\
w(1,2) & = & p(2) + q(2) + w(1,1) \quad = \quad 7 \\
c(1,2) & = & w(1,2) + \min\ \{c(1,1) + c(2,2)\} \quad = \quad 7 \\
r(0,2) & = & 2 \\
w(2,3) & = & p(3) + q(3) + w(2,2) \quad = \quad 3 \\
c(2,3) & = & w(2,3) + \min\ \{c(2,2) + c(3,3)\} \quad = \quad 3 \\
r(2,3) & = & 3 \\
w(3,4) & = & p(4) + q(4) + w(3,3) \quad = \quad 3 \\
c(3,4) & = & w(3,4) + \min\ \{c(3,3) + c(4,4)\} \quad = \quad 3 \\
r(3,4) & = & 4
\end{array}
$$

Knowing $w(i,i+1)$ and $c(i,i+1)$, $0 \le i < 4$, we can again use Equation 5.12 to compute $w(i,i+2)$, $c(i,i+2)$, and $r(i,i+2)$, $0 \le i < 3$. This process can be repeated until $w(0,4)$, $c(0,4)$, and $r(0,4)$ are obtained. The table of Figure 5.16 shows the results of this computation. The box in row i and column j shows the values of $w(j,j+i)$, $c(j,j+i)$ and $r(j,j+i)$ respectively.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 5.16 Computation of $c(0,4)$, $w(0,4)$, and $r(0,4)$

```
1   Algorithm OBST(p, q, n)
2   // Given n distinct identifiers a_1 < a_2 < ··· < a_n and probabilities
3   // p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
4   // the cost c[i, j] of optimal binary search trees t_ij for identifiers
5   // a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
6   // w[i, j] is the weight of t_ij.
7   {
8       for i := 0 to n − 1 do
9       {
10          // Initialize.
11          w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
12          // Optimal trees with one node
13          w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
14          r[i, i + 1] := i + 1;
15          c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
16      }
17      w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
18      for m := 2 to n do  // Find optimal trees with m nodes.
19          for i := 0 to n − m do
20          {
21              j := i + m;
22              w[i, j] := w[i, j − 1] + p[j] + q[j];
23              // Solve 5.12 using Knuth's result.
24              k := Find(c, r, i, j);
25                  // A value of l in the range r[i, j − 1] ≤ l
26                  // ≤ r[i + 1, j] that minimizes c[i, l − 1] + c[l, j];
27              c[i, j] := w[i, j] + c[i, k − 1] + c[k, j];
28              r[i, j] := k;
29          }
30      write (c[0, n], w[0, n], r[0, n]);
31  }
```

```
1   Algorithm Find(c, r, i, j)
2   {
3       min := ∞;
4       for m := r[i, j − 1] to r[i + 1, j] do
5           if (c[i, m − 1] + c[m, j]) < min then
6           {
7               min := c[i, m − 1] + c[m, j]; l := m;
8           }
9       return l;
10  }
```

Algorithm 5.5 Finding a minimum-cost binary search tree

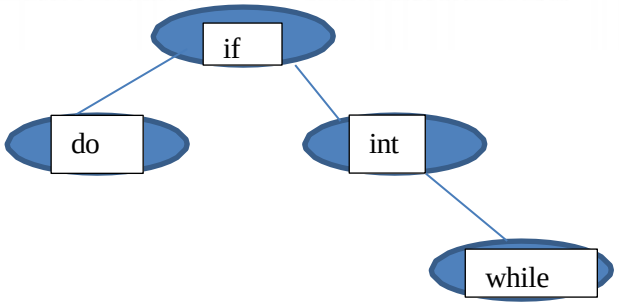


Fig: Final optimal binary search tree

0/1 – KNAPSACK:

We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x_1, x_2, x_n. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order x_n, x_{n-1}, x_1. Following a decision on x_n,

we may be in one of two possible states: the capacity remaining in m – w_n and a profit of p_n has accrued. It is clear that the remaining decisions x_{n-1}, , x_1 must be optimal with respect to the problem state resulting from the decision on x_n.Otherwise, x_n, , x_1 will not be optimal. Hence, the principal of optimality holds.

Example :

Given n=3 profit pi=(p1,p2,p3)=(1,2,5) weight wi=(w1,w2,w3)=(2,3,4) and m=6

Solution:
For the given data we have:

(P1,W1)=(1,2) (P2,W2)=(2,3)
(P3,W4)=(5,4)

For i=1
S^0= {(0,0)};
S_1^1= {(0+1, (0+2)}={(1,2)}
S^1 = (S^0 U S_1^1) = {(0, 0), (1, 2)}

for i=2

S^1 = (S^0 U S_1^1) = {(0, 0), (1, 2)}

S_1^2 ={(0+2, 0+3), (1+2, 2+3)}
={(2, 3), (3, 5)}
S^2 = (S^1 U S_1^2) = {(0, 0), (1, 2), (2, 3), (3, 5)}

for i=2
S^2 ={(0, 0), (1, 2), (2, 3), (3, 5)}

S_1^3={(0+5,0+4),(1+5,2+4),(2+5,3+4),(3+5,5+4)}={(5, 4), (6, 6), (7, 7), (8, 9)}

$S^3 = (S^2 \cup S_1^3)$ = {(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)}
By applying Dominance rule,
$S^3 = (S^2 \cup S^2)$ = {(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)}

(6,6)€S^3
(6,6)-(5,4) €S^2 then x3=1
(1,2)-(2,3) not€S^1 then x2=0
(1,2)-(1,2) €S0 then x1=1

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$
Profit $\sum$pi xi=p1*x1+p2*x2+p3*x3=6

```
1  Algorithm DKP(p, w, n, m)
2  {
3      S^0 := {(0,0)};
4      for i := 1 to n − 1 do
5      {
6          S_1^{i−1} := {(P,W)|(P − p_i, W − w_i) ∈ S^{i−1} and W ≤ m};
7          S^i := MergePurge(S^{i−1}, S_1^{i−1});
8      }
9      (PX, WX) :=last pair in S^{n−1};
10     (PY, WY) := (P' + p_n, W' + w_n) where W' is the largest W in
11         any pair in S^{n−1} such that W + w_n ≤ m;
12     // Trace back for x_n, x_{n−1}, ..., x_1.
13     if (PX > PY) then x_n := 0;
14     else x_n := 1;
15     TraceBackFor(x_{n−1}, ..., x_1);
16 }
```

Fig: Informal knapsack algoritm

String Editing:

We are given two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$, where x_i, $1 \leq i \leq n$, and y_j, $1 \leq j \leq m$, are members of a finite set of symbols known as the *alphabet*. We want to transform X into Y using a sequence of *edit operations* on X. The permissible edit operations are insert, delete, and change (a symbol of X into another), and there is a cost associated with performing each. The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence. The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y.

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_j)$ be the cost of inserting the symbol y_j into X, and $C(x_i, y_j)$ be the cost of changing the symbol x_i of X into y_j.

Example 5.19 Consider the sequences $X = x_1, x_2, x_3, x_4, x_5 = a, a, b, a, b$ and $Y = y_1, y_2, y_3, y_4 = b, a, b, b$. Let the cost associated with each insertion and deletion be 1 (for any symbol). Also let the cost of changing any symbol to any other symbol be 2. One possible way of transforming X into Y is delete each $x_i, 1 \leq i \leq 5$, and insert each $y_j, 1 \leq j \leq 4$. The total cost of this edit sequence is 9. Another possible edit sequence is delete x_1 and x_2 and insert y_4 at the end of string X. The total cost is only 3. □

the principle of optimality holds for this problem. A dynamic programming solution for this problem can be obtained as follows. Define $cost(i, j)$ to be the minimum cost of any edit sequence for transforming $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $0 \leq i \leq n$ and $0 \leq j \leq m$). Compute $cost(i, j)$ for each i and j. Then $cost(n, m)$ is the cost of an optimal edit sequence.

For $i = j = 0$, $cost(i, j) = 0$, since the two sequences are identical (and empty). Also, if $j = 0$ and $i > 0$, we can transform X into Y by a sequence of deletes. Thus, $cost(i, 0) = cost(i-1, 0) + D(x_i)$. Similarly, if $i = 0$ and $j > 0$, we get $cost(0, j) = cost(0, j-1) + I(y_j)$. If $i \neq 0$ and $j \neq 0$, $x_1, x_2, \ldots, x_i$ can be transformed into $y_1, y_2, \ldots, y_j$ in one of three ways:

1. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_j$ using a minimum-cost edit sequence and then delete x_i. The corresponding cost is $cost(i-1, j) + D(x_i)$.
2. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then change the symbol x_i to y_j. The associated cost is $cost(i-1, j-1) + C(x_i, y_j)$.
3. Transform $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then insert y_j. This corresponds to a cost of $cost(i, j-1) + I(y_j)$.

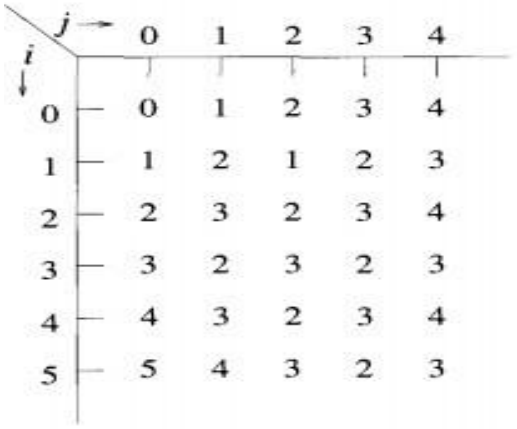
Example 5.20 Consider the string editing problem of Example 5.19. $X = a, a, b, a, b$ and $Y = b, a, b, b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i = 0, j > 1$, and $j = 0, i > 1$, $cost(i, j)$ can be computed first (Figure 5.18). Let us compute the rest of the entries in row-major order. The next entry to be computed is $cost(1, 1)$.

$$\begin{aligned} cost(1,1) &= \min\{cost(0,1) + D(x_1), cost(0,0) + C(x_1, y_1), cost(1,0) + I(y_1)\} \\ &= \min\{2, 2, 2\} = 2 \end{aligned}$$

Next is computed $cost(1, 2)$.

$$\begin{aligned} cost(1,2) &= \min\{cost(0,2) + D(x_1), cost(0,1) + C(x_1, y_2), cost(1,1) + I(y_2)\} \\ &= \min\{3, 1, 3\} = 1 \end{aligned}$$

The rest of the entries are computed similarly. Figure 5.18 displays the whole table. The value $cost(5, 4) = 3$. One possible minimum-cost edit sequence is delete x_1, delete x_2, and insert y_4. Another possible minimum cost edit sequence is change x_1 to y_2 and delete x_4. □



i \ j	0	1	2	3	4
0	0	1	2	3	4
1	1	2	1	2	3
2	2	3	2	3	4
3	3	2	3	2	3
4	4	3	2	3	4
5	5	4	3	2	3

Fig: Cost Table for above example

The Travelling Salesperson Problem:

Let G=(V,E)

be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A *tour* of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The *traveling salesperson problem* is to find a tour of minimum cost.

The traveling salesperson problem finds application in a variety of situations. Suppose we have to route a postal van to pick up mail from mail

boxes located at n different sites. An $n + 1$ vertex graph can be used to represent the situation. One vertex represents the post office from which the postal van starts and to which it must return. Edge $\langle i, j \rangle$ is assigned a cost equal to the distance from site i to site j. The route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.

In the following discussion we shall, without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1, k\}$. Hence, the principle of optimality holds. Let $g(i, S)$ be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function $g(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\}$$

Equation-1

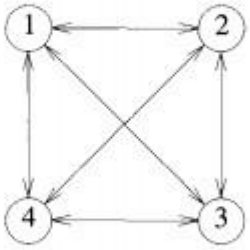
Generalization of Equation-1

we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

Equation-2

Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Fig: directed graph and edge length matrix c

Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

$$\begin{array}{lllll} g(2,\{3\}) & = & c_{23} + g(3,\phi) & = & 15 \\ g(3,\{2\}) & = & 18 & & \\ g(4,\{2\}) & = & 13 & & \end{array} \qquad \begin{array}{lll} g(2,\{4\}) & = & 18 \\ g(3,\{4\}) & = & 20 \\ g(4,\{3\}) & = & 15 \end{array}$$

Next, we compute $g(i,S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lllll} g(2,\{3,4\}) & = & \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} & = & 25 \\ g(3,\{2,4\}) & = & \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} & = & 25 \\ g(4,\{2,3\}) & = & \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} & = & 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{array}{lll} g(1,\{2,3,4\}) & = & \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ & = & \min\{35, 40, 43\} \\ & = & 35 \end{array}$$

An optimal tour of the graph of Figure 5.21(a) has length 35. A tour of this length can be constructed if we retain with each $g(i,S)$ the value of J that minimizes right hand side of equation-2.Let J(i,S) be this value. Then, $J(1,\{2,3,4\}) = 2$. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from $g(2, \{3, 4\})$. So $J(2, \{3, 4\}) = 4$. Thus the next edge is $\langle 2,4\rangle$. The remaining tour is for $g(4, \{3\})$. So $J(4, \{3\}) = 3$. The optimal tour is 1, 2, 4, 3, 1. □

Single Source Shortest Path (General Weights) – Bellman-Ford Algorithm

Introduction
- SSSP finds the minimum-cost path from one source vertex to every other vertex.
- Applications: GPS, routing, computer networks, robotics, logistics.

.Prerequisites
- Weighted graphs
- Directed and undirected graphs
- Positive and negative edge weights

What is SSSP?
- Given G(V,E) and a source vertex s, compute the shortest distance from s to every vertex.

Why Bellman-Ford?
- Dijkstra works only for non-negative edge weights.
- Bellman-Ford supports negative edge weights and detects negative-weight cycles.

Key Concept: Edge Relaxation
- For edge (u,v) with weight w: if dist[u]+w < dist[v], update dist[v].

Algorithm Steps
- Initialize source distance = 0 and all others = infinity.
- Repeat relaxation of all edges V-1 times.
- Perform one extra pass; if any distance decreases, a negative cycle exists.

Worked Example
- Use a graph with 4-5 vertices and show distance updates after each iteration.

Pseudocode
- Initialize distances.
- Repeat V-1 times: relax every edge.
- Check for negative cycle.

Complexity
- Time: O(VE)
- Space: O(V)

Advantages
- Handles negative weights.
- Detects negative cycles.
- Simple to implement.

Limitations
- Slower than Dijkstra.
- Not efficient for very large dense graphs.

Applications
- Distance-vector routing protocols.
- Currency arbitrage detection.
- Transportation and network optimization.

Bellman-Ford vs Dijkstra

- Bellman-Ford: negative weights supported, O(VE).
- Dijkstra: non-negative weights only, faster with priority queue.

Short Answer Questions

1. Define a Greedy Algorithm. Mention its characteristics.
2. Differentiate between a feasible solution and an optimal solution.
3. State the Greedy strategy used in the Fractional Knapsack problem.
4. What is the objective of the Job Sequencing with Deadlines problem?
5. Define a Minimum Spanning Tree (MST).
6. List any two applications of Minimum Spanning Trees.
7. State the principle used in Kruskal's Algorithm.
8. What is the basic idea behind Prim's Algorithm?
9. Define Single Source Shortest Path Problem.
10. Why is Dijkstra's Algorithm considered a Greedy algorithm?
11. Define Dynamic Programming.
12. Differentiate between Greedy Method and Dynamic Programming.
13. What is Floyd's Algorithm used for?
14. Define an Optimal Binary Search Tree (OBST).
15. Differentiate between Fractional Knapsack and 0/1 Knapsack.
16. What is String Editing? Mention one application.
17. Define Travelling Salesperson Problem (TSP).
18. State the Principle of Optimality in Dynamic Programming.
19. Mention the time complexity of Floyd's Algorithm.
20. List any two real-world applications of Dijkstra's Algorithm.

Long Answer Questions

1. Explain the Greedy Method with its control abstraction algorithm. Illustrate the concepts of feasible and optimal solutions with a suitable example.
2. Solve the Fractional Knapsack problem using the Greedy approach for a given set of objects. Explain the algorithm and analyze its time complexity.
3. Explain the Job Sequencing with Deadlines algorithm using a suitable example and discuss its time complexity.
4. Explain Kruskal's Algorithm to construct a Minimum Cost Spanning Tree with a suitable example. Analyze its time complexity.
5. Explain Prim's Algorithm with an example. Compare Prim's and Kruskal's algorithms.
6. Explain Dijkstra's Algorithm for finding the Single Source Shortest Path. Demonstrate the algorithm using a weighted graph and analyze its complexity.
7. Explain the Dynamic Programming approach. Compare Dynamic Programming with the Greedy Method using suitable examples.
8. Explain Floyd's Algorithm for solving the All-Pairs Shortest Path problem with a suitable example and complexity analysis.
9. Explain the construction of an Optimal Binary Search Tree (OBST). Discuss its advantages and applications.
10. Solve the 0/1 Knapsack problem using Dynamic Programming and explain the Principle of Optimality.
11. Explain the String Editing problem using Dynamic Programming with a suitable example.
12. Explain the Dynamic Programming solution for the Travelling Salesperson Problem (TSP) with a suitable example and discuss its time complexity.

Case Study – Unit III

Smart Logistics and E-Commerce Delivery Optimization Using Greedy and Dynamic Programming Algorithms

Course

Algorithms, Data Structures and Algorithm Analysis (ADS&AA)

Difficulty Level

Intermediate

Learning Outcomes

After completing this case study, students will be able to:

- Apply Greedy algorithms to optimization problems.
- Solve resource allocation problems using Knapsack algorithms.
- Schedule jobs efficiently using Job Sequencing.
- Design minimum-cost communication networks using MST algorithms.
- Determine shortest routes using Dijkstra's and Floyd's algorithms.
- Apply Dynamic Programming for optimization problems.
- Design efficient search structures using Optimal Binary Search Trees.
- Analyze real-world routing problems using the Travelling Salesperson Problem.

Background

QuickKart, a nationwide e-commerce company, delivers products to more than **500 cities** using:

- 50 regional warehouses
- 120 delivery hubs
- 300 delivery vehicles
- Thousands of daily customer orders

The company aims to:

- Deliver products faster.
- Minimize transportation cost.
- Optimize warehouse connectivity.
- Schedule deliveries efficiently.
- Improve customer search performance.
- Reduce fuel consumption.

Problem Statement

QuickKart requires an intelligent logistics system that can:

- Select high-value packages for delivery vehicles.
- Schedule urgent deliveries first.
- Connect warehouses at minimum cost.

- Determine shortest delivery routes.
- Find shortest routes between all cities.
- Optimize frequently searched products.
- Minimize editing effort in customer records.
- Determine the most efficient delivery tour.

Proposed Solution

Part A – Greedy Method

The logistics system follows the Greedy strategy by making the **best local choice at each step** to obtain an optimal or near-optimal solution. It is suitable for scheduling, routing, and resource allocation problems.

Part B – Fractional Knapsack

Each delivery truck has a limited weight capacity.

Packages have:

- Weight
- Profit (delivery value)

The system selects packages based on the **highest profit-to-weight ratio** until the vehicle reaches capacity, allowing fractional loading when appropriate.

Example

Truck Capacity = **100 kg**

Package	Weight	Profit	Profit/Weight
A	20	₹4000	200
B	30	₹4500	150
C	50	₹6000	120

The truck first selects the package with the highest profit-to-weight ratio to maximize revenue.

Part C – Job Sequencing with Deadlines

Thousands of delivery requests arrive daily.

Each order contains:

- Deadline
- Profit
- Delivery duration

The system schedules the most profitable deliveries before their deadlines, maximizing total revenue while respecting time constraints.

Part D – Minimum Cost Spanning Tree (MST)

QuickKart plans to connect warehouses using fiber-optic communication links.

Objectives:

- Connect all warehouses.
- Avoid communication loops.
- Minimize installation cost.

Kruskal's Algorithm

Edges are selected in increasing order of cost while avoiding cycles, resulting in a minimum-cost network.

Prim's Algorithm

Starting from one warehouse, the network is expanded by repeatedly adding the cheapest connecting edge until all warehouses are connected.

Part E – Dijkstra's Algorithm

Delivery vehicles must find the shortest route from a warehouse to multiple customer locations.

The road network is represented as a weighted graph:

- Vertices → Cities
- Edges → Roads
- Weights → Distance or travel time

Dijkstra's algorithm computes the shortest path from one source warehouse to every destination with non-negative edge weights.

Part F – Dynamic Programming

Some optimization problems require evaluating many possible decision sequences.

QuickKart uses Dynamic Programming because it stores solutions to smaller subproblems and reuses them, avoiding repeated calculations. This bottom-up approach is effective for routing and optimization tasks.

Part G – Floyd's Algorithm (All-Pairs Shortest Paths)

The company frequently computes the shortest route between every pair of warehouses.

Floyd's Algorithm efficiently determines shortest paths between all city pairs, enabling route planning and traffic analysis.

Part H – Optimal Binary Search Tree (OBST)

The product database stores thousands of items.

Frequently searched products such as:

- Mobile Phones
- Laptops
- Grocery Items

are placed closer to the root of an Optimal Binary Search Tree to reduce average search time.

Part I – 0/1 Knapsack

Unlike the Fractional Knapsack, each package must be either fully selected or not selected at all.

This model is appropriate for:

- Electronic devices
- Furniture
- Fragile products

where partial delivery is impossible. Dynamic Programming identifies the optimal subset of packages within the truck's capacity.

Part J – String Editing

Customer information often contains typing errors.

Examples:

- "Hyderbad" → "Hyderabad"
- "Anush" → "Anusha"

String Editing computes the minimum insertions, deletions, and substitutions needed to correct such errors, improving data quality.

Part K – Travelling Salesperson Problem (TSP)

A delivery vehicle must:

- Start from the warehouse.
- Visit every customer exactly once.
- Return to the warehouse.
- Minimize total travel distance.

This classic optimization problem is solved using Dynamic Programming for small to medium-sized instances.

Algorithm Analysis

Algorithm	Real-World Application	Time Complexity
Greedy Method	Resource allocation	Problem-dependent
Fractional Knapsack	Truck loading	O(n log n)
Job Sequencing	Delivery scheduling	O(n log n)
Kruskal's MST	Network design	O(E log E)
Prim's MST	Infrastructure planning	O(E log V) (with efficient data structures)

Dijkstra	Single-source routing	$O((V + E) \log V)$
Floyd	All-pairs routing	$O(n^3)$
OBST	Product search optimization	$O(n^3)$
0/1 Knapsack	Package selection	$O(nW)$
String Editing	Error correction	$O(mn)$
TSP (Dynamic Programming)	Route optimization	$O(n^2 \cdot 2^n)$

Real-World Applications

These algorithms are widely used in:

- Amazon Logistics
- Flipkart Delivery Network
- Google Maps
- FedEx and DHL
- Airline Route Planning
- Banking Systems
- Hospital Supply Chains
- Warehouse Management Systems
- Navigation Systems
- Cloud Data Centers

Discussion Questions

1. Why is the Greedy Method suitable for Fractional Knapsack but not for the 0/1 Knapsack problem?
2. Explain how Job Sequencing with Deadlines increases profit in delivery scheduling.
3. Compare Kruskal's and Prim's algorithms for designing communication networks.
4. How does Dijkstra's algorithm improve logistics route planning?
5. Why is Floyd's algorithm preferred when shortest paths between all city pairs are required?
6. Explain the advantages of Optimal Binary Search Trees in e-commerce applications.
7. Differentiate Fractional Knapsack and 0/1 Knapsack using suitable examples.
8. How does Dynamic Programming improve the efficiency of the Travelling Salesperson Problem?

Conclusion

The **Smart Logistics and E-Commerce Delivery System** illustrates how **Greedy algorithms** and **Dynamic Programming** can solve practical optimization problems in transportation, warehouse management, networking, product search, and route planning. Algorithms such as **Fractional Knapsack, Job Sequencing, Kruskal's, Prim's, Dijkstra's, Floyd's Algorithm, Optimal Binary Search Trees, 0/1 Knapsack, String Editing, and the Travelling Salesperson Problem** enable organizations to reduce operational costs, improve delivery efficiency, and enhance customer satisfaction.

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, xn) where each xi Є 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 (x1, xn). 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:

1. Explicit and

2. Implicit constraints.

Definition 1: Explicit constraints are rules that restrict each to take on values only from a given set. Explicit constraints depend on the particular instance of problem being solved. All tuples that satisfy the explicit constraints

Definition 2: Implicit constraints are rules that determine which of the tuples in the

solution space of I satisfy the criterion function.

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 be on 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.

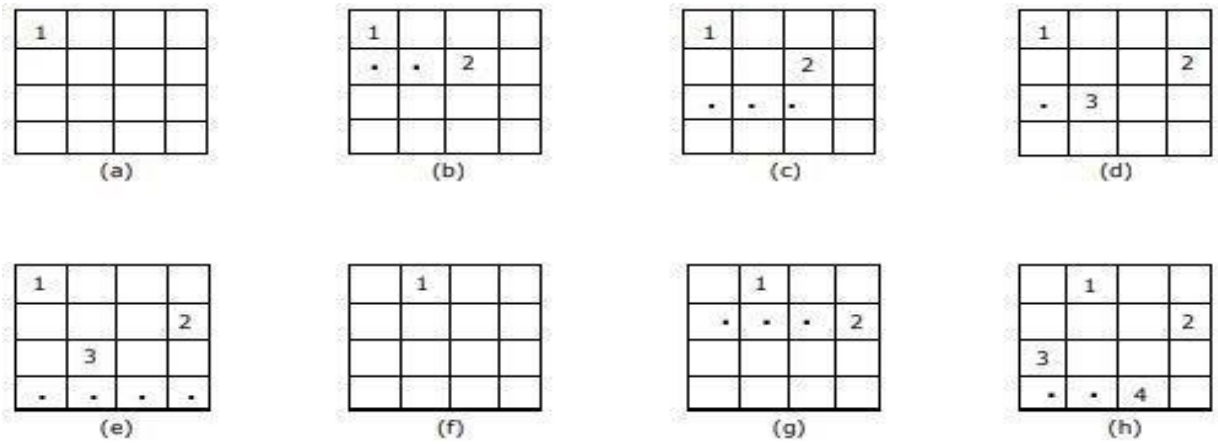
N-Queens Problem:

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 (x1, ... , x8), where xi is the column of the ith row where the ith queen is placed.

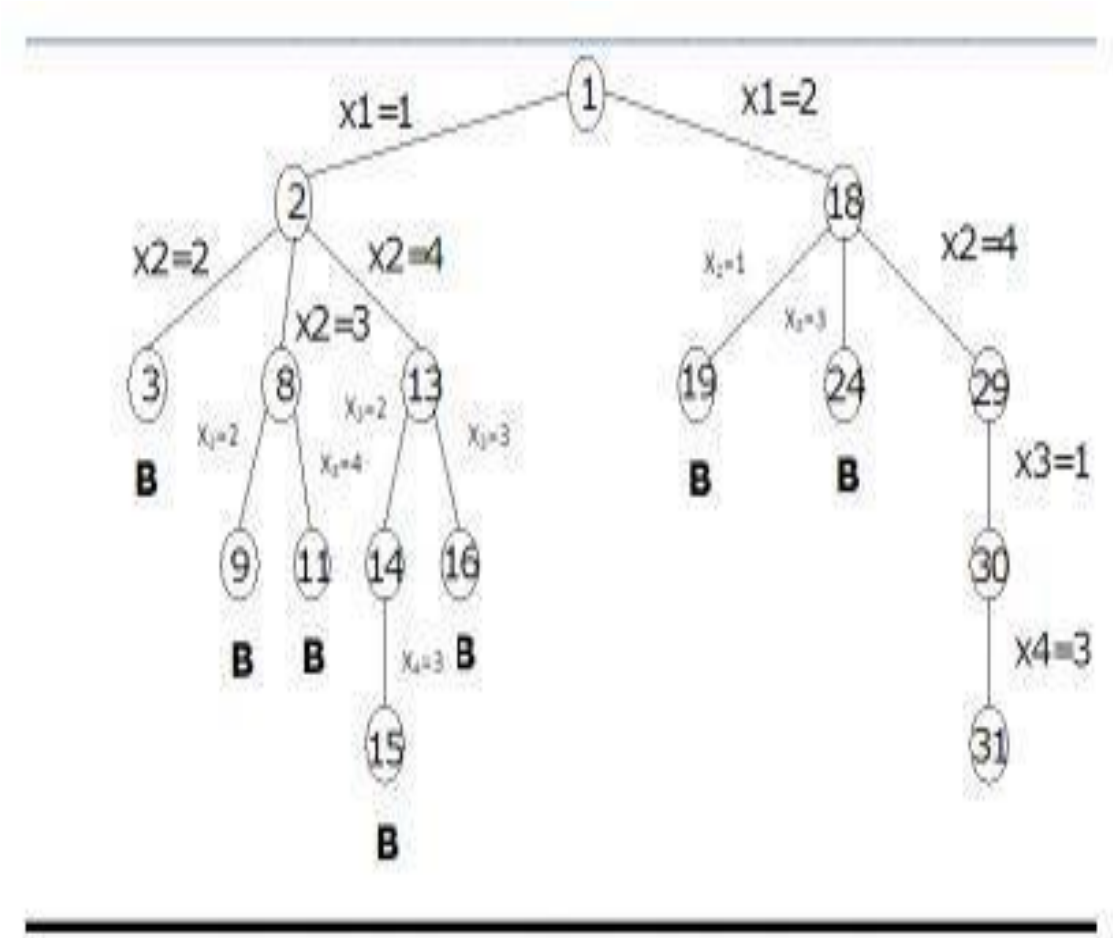
The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, 1 <i < 8. Therefore the solution space consists of 888-tuples. The implicit constraints for this problem are that no two xi"s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal. This realization reduces the size of the solution space from 88 tuples to 8! Tuples. The promising function must check whether two queens are in the same column or diagonal:

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:

$1 + n + n^2 + n^3 + \cdots + n^n$

$= \frac{n^{n+1} - 1}{n - 1}$

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

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

SUM OF SUBSETS PROBLEM:

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of wi whose sums are „m". All solutions are k-tuples, $1 \le k \le n$.

Explicit constraints:

- xi Є {j | j is an integer and $1 \le j \le n$}.

Implicit constraints:

- No two xi can be the same.
- The sum of the corresponding wi".
- $xi < xi+1$, $1 \le i < k$ (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).

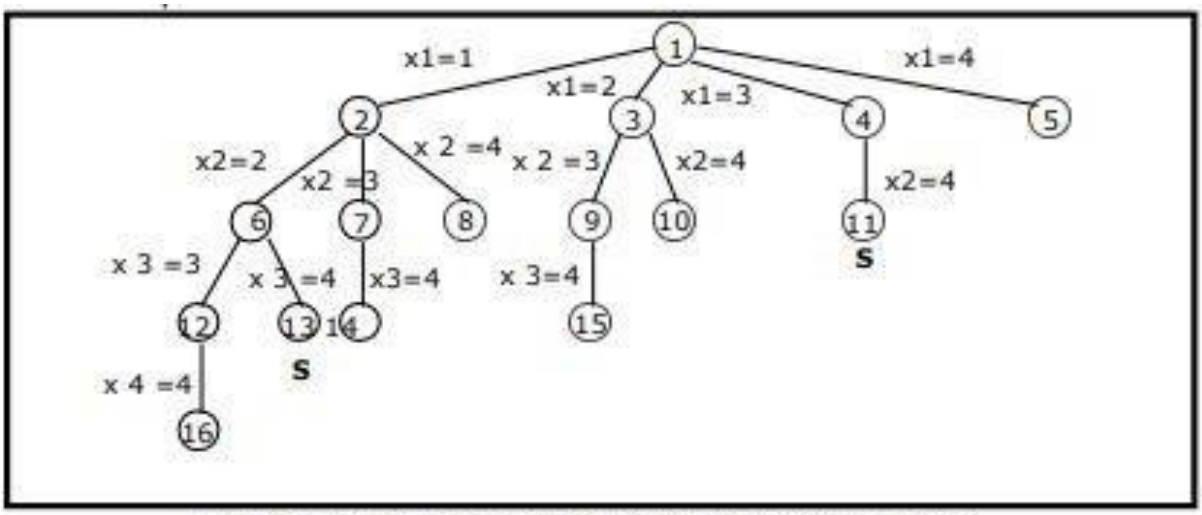
A better formulation of the problem is where the solution subset is represented by an n-tuple(x1,....., xn) such that xi Є {0,1}. The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2n distinct tuples.

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



Apossiblesolutionspaceorganisationforthe s u m of t h e s u b s et s pro blem.

The tree corresponds to the variable tuple size formulation. The edges are labelled such that an edge from a level i node to a level i+1 node represents a value for xi. 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 w1, the next sub-tree defines all subsets containing w2 but not w1, 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

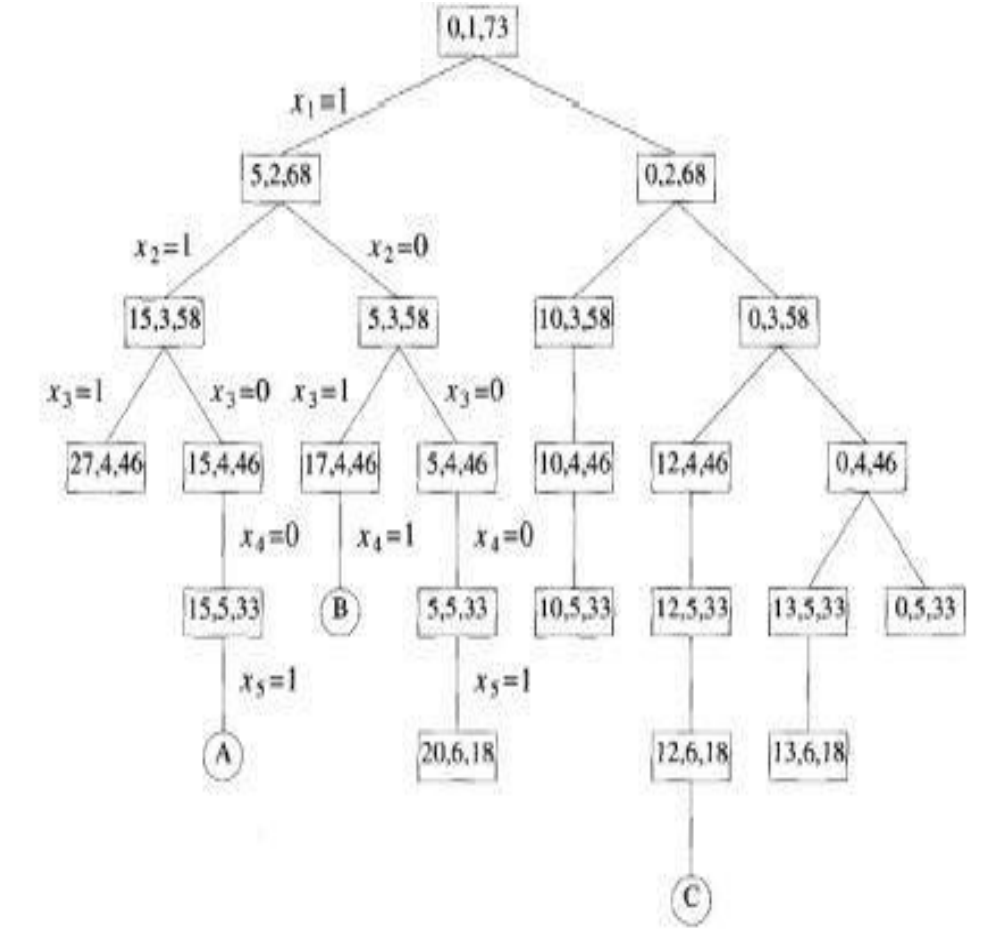


Figure 7.10 Portion of state space tree generated by SumOfSub

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 (x1, x2, . . ., xn), where xi is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement m coloring(1);

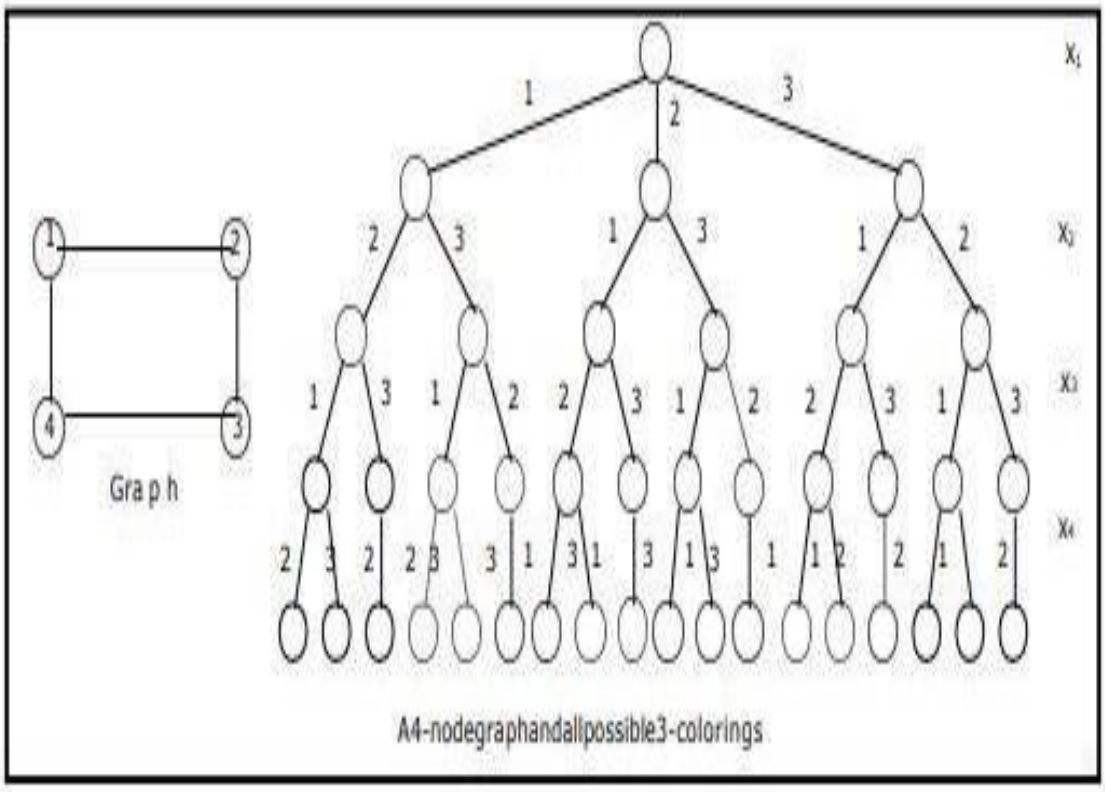
```
Algorithm mcoloring (k)
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
//1,2,.....,m to the vertices of the graph such that adjacent verticesareassigned
// distinct integers are printed. k is the index of the next vertex to color.
{
        repeat
        {                                       // Generate all legal assignments forx[k].
                NextValue (k);          // Assign to x [k] a legal color.
                If (x [k] = 0) thenreturn;      // No new colorpossible
                If (k =  n)then         // at most m colors havebeen
                                        // used to color the n vertices.
                        write (x [1: n]);
                        else mcoloring (k+1);
                } until (false);
        }
```

```
Algorithm NextValue (k)
// x[1], .....x[k-1]havebeenassignedintegervaluesintherange[1,m]suchthat
// adjacent vertices have distinct integers. A value for x [k] is determined in therange
// [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 highestcolor.
                If (x [k] = 0) thenreturn;                      // All colors have beenused
                for j := 1 to ndo
                {       // 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)thenreturn;              // New colorfound
        }until(false);                           // Otherwise try to find anothercolor.
}
```

Example:

Color the graph given below with minimum number of colors by backtracking using state space tree



0/1 KNAPSACK PROBLEM

7.6 KNAPSACK PROBLEM

In this section we reconsider a problem that was defined and solved by a dynamic programming algorithm in Chapter 5, the 0/1 knapsack optimization problem. Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized} \tag{7.2}$$

The x_i's constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's. Thus the solution space is the same as that for the sum of subsets problem. Two possible tree organizations are possible. One corresponds to the fixed tuple size formulation (Figure 7.4) and the other to the variable tuple size formulation (Figure 7.3). Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees. Regardless of which is used, bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than

the value of the best solution determined so far, then that live node can be killed.

We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i, $1 \le i \le k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \le x_i \le 1$ for $k+1 \le i \le n$ and using the greedy algorithm of Section 4.2 to solve the relaxed problem. Function Bound(cp, cw, k) (Algorithm 7.11) determines an upper bound on the best solution obtainable by expanding any node Z at level $k+1$ of the state space tree. The object weights and profits are $w[i]$ and $p[i]$. It is assumed that $p[i]/w[i] \ge p[i+1]/w[i+1]$, $1 \le i < n$.

```
1   Algorithm Bound(cp, cw, k)
2   // cp is the current profit total, cw is the current
3   // weight total; k is the index of the last removed
4   // item; and m is the knapsack size.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           c := c + w[i];
9           if (c < m) then b := b + p[i];
10          else return b + (1 - (c - m)/w[i]) * p[i];
11      }
12      return b;
13  }
```

Algorithm 7.11 A bounding function

From Bound it follows that the bound for a feasible left child of a node Z is the same as that for Z. Hence, the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node. The resulting algorithm is BKnap (Algorithm 7.12). It was obtained from the recursive backtracking schema. Initially set $fp := -1$;. This algorithm is invoked as

```
BKnap(1, 0, 0);
```

When $fp \ne -1$, $x[i]$, $1 \le i \le n$, is such that $\sum_{i=1}^{n} p[i]x[i] = fp$. In lines 8 to 18 left children are generated. In line 20, Bound is used to test whether a

```
1   Algorithm BKnap(k, cp, cw)
2   // m is the size of the knapsack; n is the number of weights
3   // and profits. w[ ] and p[ ] are the weights and profits.
4   // p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
5   // knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
6   // is not in the knapsack; else x[k] = 1.
7   {
8       // Generate left child.
9       if (cw + w[k] ≤ m) then
10      {
11          y[k] := 1;
12          if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
13          if ((cp + p[k] > fp) and (k = n)) then
14          {
15              fp := cp + p[k]; fw := cw + w[k];
16              for j := 1 to k do x[j] := y[j];
17          }
18      }
19      // Generate right child.
20      if (Bound(cp, cw, k) ≥ fp) then
21      {
22          y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
23          if ((cp > fp) and (k = n)) then
24          {
25              fp := cp; fw := cw;
26              for j := 1 to k do x[j] := y[j];
27          }
28      }
29  }
```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

right child should be generated. The path $y[i]$, $1 \le i \le k$, is the path to the current node. The current weight $cw = \sum_{i=1}^{k-1} w[i]y[i]$ and $cp = \sum_{i=1}^{k-1} p[i]y[i]$. In lines 13 to 17 and 23 to 27 the solution vector is updated if need be.

So far, all our backtracking algorithms have worked on a static state space tree. We now see how a dynamic state space tree can be used for the knapsack problem. One method for dynamically partitioning the solution space is based on trying to obtain an optimal solution using the greedy algorithm of Section 4.2. We first replace the integer constraint $x_i = 0$ or 1 by the constraint $0 \le x_i \le 1$. This yields the relaxed problem

$$\max \sum_{1 \le i \le n} p_i x_i \quad \text{subject to} \quad \sum_{1 \le i \le n} w_i x_i \le m \tag{7.3}$$

$$0 \le x_i \le 1, \qquad 1 \le i \le n$$

If the solution generated by the greedy method has all x_i's equal to zero or one, then it is also an optimal solution to the original 0/1 knapsack problem. If this is not the case, then exactly one x_i will be such that $0 < x_i < 1$. We partition the solution space of (7.2) into two subspaces. In one $x_i = 0$ and in the other $x_i = 1$. Thus the left subtree of the state space tree will correspond to $x_i = 0$ and the right to $x_i = 1$. In general, at each node Z of the state space tree the greedy algorithm is used to solve (7.3) under the added restrictions corresponding to the assignments already made along the path from the root to this node. In case the solution is all integer, then an optimal solution for this node has been found. If not, then there is exactly one x_i such that $0 < x_i < 1$. The left child of Z corresponds to $x_i = 0$, and the right to $x_i = 1$.

The justification for this partitioning scheme is that the noninteger x_i is what prevents the greedy solution from being a feasible solution to the 0/1 knapsack problem. So, we would expect to reach a feasible greedy solution quickly by forcing this x_i to be integer. Choosing left branches to correspond to $x_i = 0$ rather than $x_i = 1$ is also justifiable. Since the greedy algorithm requires $p_j/w_j \ge p_{j+1}/w_{j+1}$, we would expect most objects with low index (i.e., small j and hence high density) to be in an optimal filling of the knapsack. When x_i is set to zero, we are not preventing the greedy algorithm from using any of the objects with $j < i$ (unless x_j has already been set to zero). On the other hand, when x_i is set to one, some of the x_j's with $j < i$ will not be able to get into the knapsack. Therefore we expect to arrive at an optimal solution with $x_i = 0$. So we wish the backtracking algorithm to try this alternative first. Hence the left subtree corresponds to $x_i = 0$.

Example 7.7 Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: $p = \{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. The greedy

solution corresponding to the root node (i.e., Equation (7.3)) is $x = \{1, 1, 1, 1, 1, 21/45, 0, 0\}$. Its value is 164.88. The two subtrees of the root correspond to $x_6 = 0$ and $x_6 = 1$, respectively (Figure 7.16). The greedy solution at node 2 is $x = \{1, 1, 1, 1, 1, 0, 21/45, 0\}$. Its value is 164.66. The solution space at node 2 is partitioned using $x_7 = 0$ and $x_7 = 1$. The next E-node is node 3. The solution here has $x_8 = 21/55$. The partitioning now is with $x_8 = 0$ and $x_8 = 1$. The solution at node 4 is all integer so there is no need to expand this node further. The best solution found so far has value 139 and $x = \{1, 1, 1, 1, 1, 1, 0, 0, 0\}$. Node 5 is the next E-node. The greedy solution for this node is $x = \{1, 1, 1, 22/23, 0, 0, 0, 1\}$. Its value is 159.56. The partitioning is now with $x_4 = 0$ and $x_4 = 1$. The greedy solution at node 6 has value 156.66 and $x_5 = 2/3$. Next, node 7 becomes the E-node. The solution here is $\{1, 1, 1, 0, 0, 0, 0, 1\}$. Its value is 128. Node 7 is not expanded as the greedy solution here is all integer. At node 8 the greedy solution has value 157.71 and $x_3 = 4/7$. The solution at node 9 is all integer and has value 140. The greedy solution at node 10 is $\{1, 0, 1, 0, 1, 0, 0, 1\}$. Its value is 150. The next E-node is 11. Its value is 159.52 and $x_3 = 20/21$. The partitioning is now on $x_3 = 0$ and $x_3 = 1$. The remainder of the backtracking process on this knapsack instance is left as an exercise. □

Experimental work due to E. Horowitz and S. Sahni, cited in the references, indicates that backtracking algorithms for the knapsack problem generally work in less time when using a static tree than when using a dynamic tree. The dynamic partitioning scheme is, however, useful in the solution of integer linear programs. The general integer linear program is mathematically stated in (7.4).

$$\begin{aligned} &\text{minimize} \quad \textstyle\sum_{1 \le j \le n} c_j x_j \\ &\text{subject to} \quad \textstyle\sum_{1 \le j \le n} a_{ij} x_j \le b_i, \quad 1 \le i \le m \qquad (7.4) \\ &\qquad x_j's \text{ are nonnegative integers} \end{aligned}$$

If the integer constraints on the x_i's in (7.4) are replaced by the constraint $x_i \ge 0$, then we obtain a linear program whose optimal solution has a value at least as large as the value of an optimal solution to (7.4). Linear programs can be solved using the simplex methods (see the references). If the solution is not all integer, then a noninteger x_i is chosen to partition the solution space. Let us assume that the value of x_i in the optimal solution to the linear program corresponding to any node Z in the state space is v and v is not an integer. The left child of Z corresponds to $x_i \le \lfloor v \rfloor$ whereas the right child of Z correspond to $x_i \ge \lceil v \rceil$. Since the resulting state space tree has a potentially infinite depth (note that on the path from the root to a node Z

the solution space can be partitioned on one x_i many times as each x_i can have as value any nonnegative integer), it is almost always searched using a branch-and-bound method (see Chapter 8).

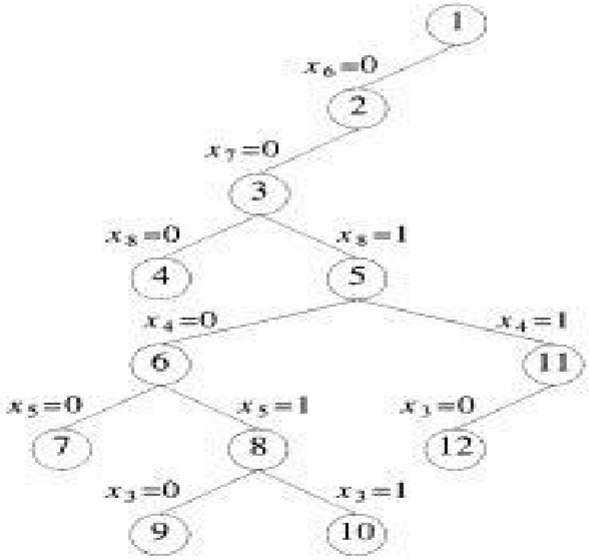


Figure 7.16 Part of the dynamic state space tree generated in Example 7.7

Branch and Bound

Introduction:

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.
- A D-search like state space search is called as LIFO (last in first out) search as the list of live nodes in a last in first out list.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is

a node currently being expanded.

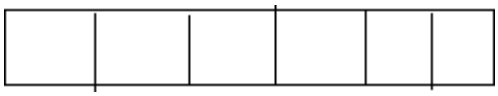
Dead node is a generated anode that is not be expanded or explored any further. All children of a dead node have already been expanded.

Here we will use 3 types of search strategies:

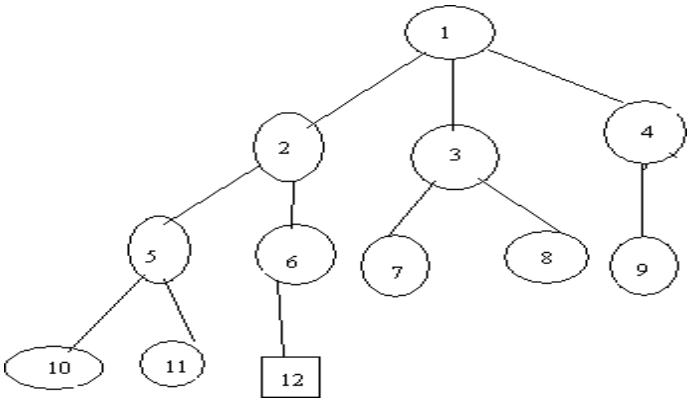
1. FIFO (First In First Out)
2. LIFO (Last In First Out)
3. LC (Least Cost) Search

FIFO Branch and Bound Search:

For this we will use a data structure called Queue. Initially Queue is empty.



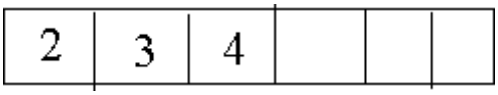
Example:



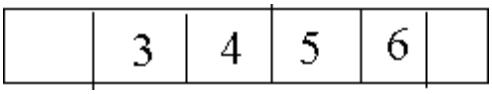
Assume the node 12 is an answer node (solution)

In FIFO search, first we will take E-node as a node 1.

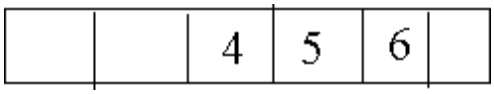
Next we generate the children of node 1. We will place all these live nodes in a queue.



Now we will delete an element from queue, i.e. node 2, next generate children of node 2 and place in this queue.



Next, delete an element from queue and take it as E-node, generate the children of node 3, 7, 8 are children of 3 and these live nodes are killed by bounding functions. So we will not include in the queue.



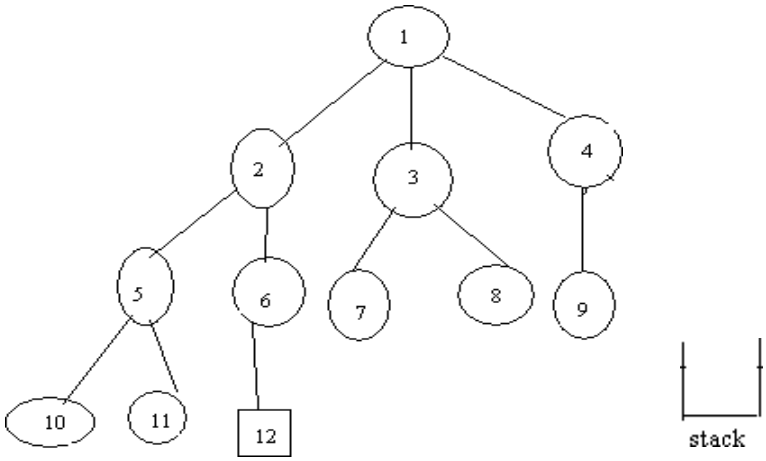
Again delete an element an from queue. Take it as E-node, generate the children of 4. Node 9 is generated and killed by boundary function.

Next, delete an element from queue. Generate children of nodes 5, i.e., nodes 10 and 11 are generated and by boundary function, last node in queue is 6. The child of node 6 is 12 and it satisfies the conditions of the problem, which is the answer node, so search terminates.

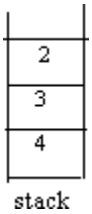
LIFO Branch and Bound Search

For this we will use a data structure called stack. Initially stack is empty.

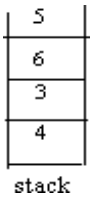
Example:



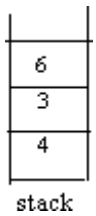
Generate children of node 1 and place these live nodes into stack.



Remove element from stack and generate the children of it, place those nodes into stack. 2 is removed from stack. The children of 2 are 5, 6. The content of stack is,



Again remove an element from stack, i.,e node 5 is removed and nodes generated by 5 are 10, 11 which are killed by bounded function, so we will not place 10, 11 into stack.

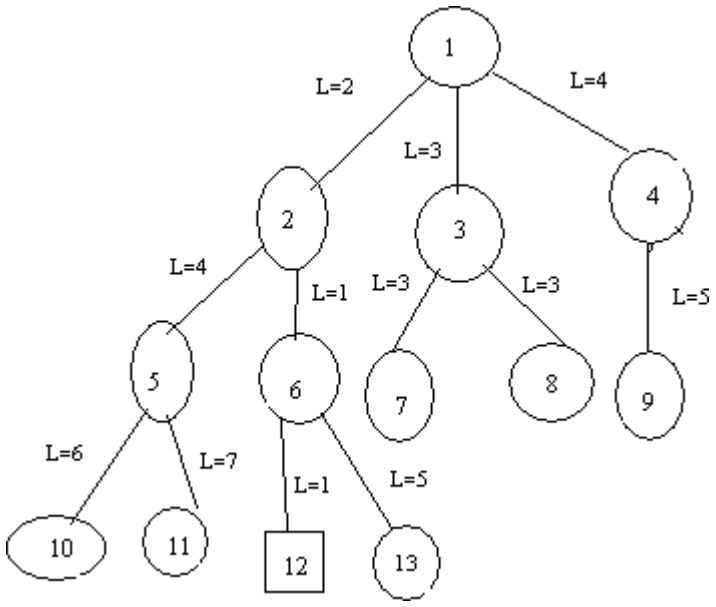


Delete an element from stack, i.,e node 6. Generate child of node 6, i.,e 12, which is the answer node, so search process terminates.

LC (Least Cost) Branch and Bound Search

In both FIFO and LIFO Branch and Bound the selection rules for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preferences to a node that has a very good chance of getting the search to an answer node quickly.

In this we will use ranking function or cost function. We generate the children of E-node, among these live nodes; we select a node which has minimum cost. By using ranking function we will calculate the cost of each node.



Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4. By using ranking function we will calculate the cost of 2, 3, 4 nodes is $\hat{c}$ =2, $\hat{c}$ =3, $\hat{c}$ =4 respectively. Now we will select a node which has minimum cost i.,e node 2. For node 2, the children are 5, 6. Between 5 and 6 we will select the node 6 since its cost minimum. Generate children of node 6 i.,e 12 and 13. We will select node 12 since its cost ($\hat{c}$ =1) is minimum. More over 12 is the answer node. So, we terminate search process.

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 sub tree with root x. Thus, c(t) is the cost of a

minimum-cost answer node in t.

LC search uses ĉ to find an answer node. The algorithm uses two functions

1. Least-cost()
2. Add_node().

Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

Add_node() to delete and add a live node from or to the list of live nodes.

Add_node(x)adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

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.
- A good bounding helps to prune (reduce) efficiently the tree, leading to a faster exploration of the solution space. Each time a new answer node is found, the value of upper can be updated.
- Branch and bound algorithms are used for optimization problem 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.

0/1 KNAPSACK PROBLEM (LCBB)

There are n objects given and capacity of knapsack is M. Select some objects to fill the knapsack in such a way that it should not exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is maximization problem. It means we will always seek for maximum p_1x_1 (where p_1 represents profit of object x_1).

A branch bound technique is used to find solution to the knapsack problem. But we cannot directly apply the branch and bound technique to the knapsack problem. Because the branch bound deals only the minimization problems. We modify the knapsack problem to the minimization problem. The modifies problem is,

$$\text{minimize} \quad -\sum_{i=1}^{n} p_i x_i$$

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

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

```
Algorithm Reduce(p, w, n, m, I1, I2)
// Variables are as described in the discussion.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1], 1 ≤ i < n.
{
    I1 := I2 := ∅;
    q := Lbb(∅, ∅);
    k := largest j such that w[1] + ··· + w[j] < m;
    for i := 1 to k do
    {
        if (Ubb(∅, {i}) < q) then I1 := I1 ∪ {i};
        else if (Lbb(∅, {i}) > q) then q := Lbb(∅, {i});
    }
    for i := k + 1 to n do
    {
        if (Ubb({i}, ∅) < q) then I2 := I2 ∪ {i};
        else if (Lbb({i}, ∅) > q) then q := Lbb({i}, ∅);
    }
}
```

Algorithm: KNAPSACK PROBLEM

Example: Consider the instance M=15, n=4, (p_1, p_2, p_3, p_4) = 10, 10, 12, 18 and (w_1, w_2, w_3, w_4)=(2, 4, 6, 9).

Solution: knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Arrange the item profits and weights with respect of profit by weight ratio. After that, place the first item in the knapsack. Remaining weight of knapsack is 15-2=13. Place next item w_2 in knapsack and the remaining weight of knapsack is 13-4=9. Place next item w_3, in knapsack then the remaining weight of knapsack is 9-6=3. No fraction are allowed in calculation of upper bound so w_4, cannot be placed in knapsack.

Profit= $p_1+p_2+p_3$,=10+10+12

So, Upper bound=32

To calculate Lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound=10+10+12+ (3/9*18)=32+6=38

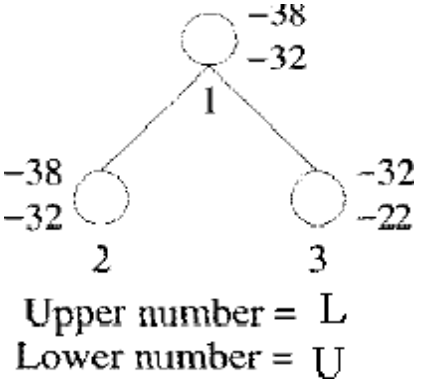
Knapsack is maximization problem but branch bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, upper bound
(U) =-32 Lower bound
(L)=-38

We choose the path, which has minimized difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.Now we will calculate upper bound and lower bound for nodes 2, 3

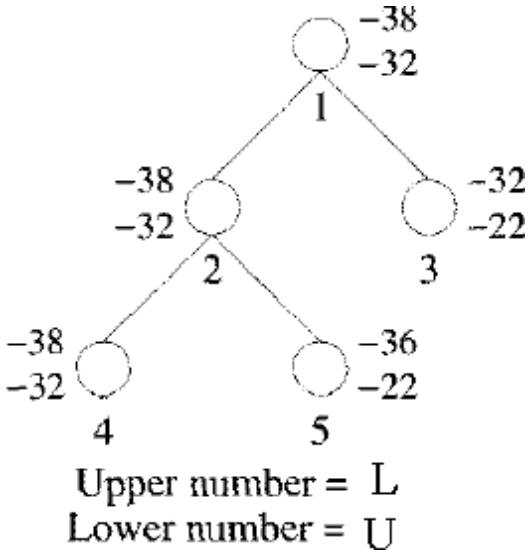
For node 2, x_1=1, means we should place first item in the knapsack.
U=10+10+12=32, make it as -32
L=10+10+12+ (3/9*18) = 32+6=38, we make it as -38

For node 3, x_1=0, means we should not place first item in the knapsack.
U=10+12=22, make it as -22
L=10+12+ (5/9*18) = 10+12+10=32, we make it as -32



Next we will calculate difference of upper bound and lower bound for
nodes 2, 3 For node 2, U-L=-32+38=6
For node 3, U-L=-22+32=10

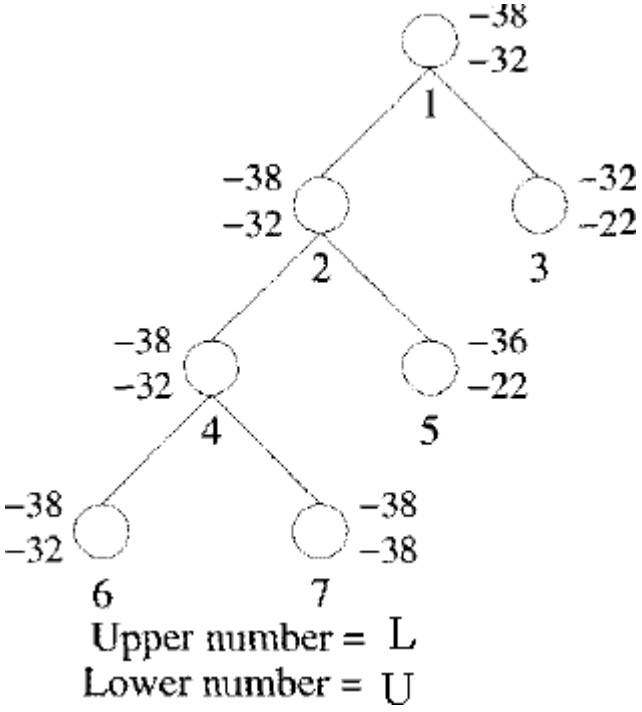
Choose node 2, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U-L=-
32+38=6 For node 5,
U-L=-22+36=14

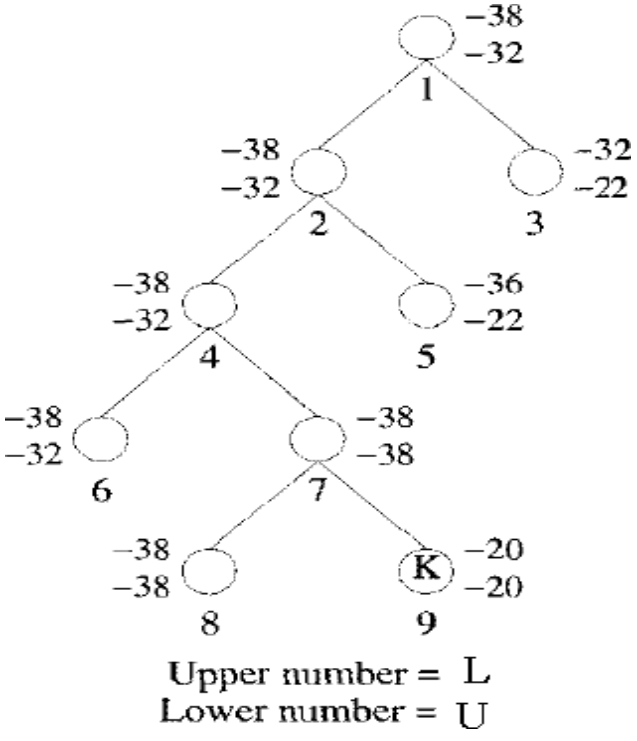
Choose node 4, since it has minimum difference value of 6



Now we will calculate lower bound and upper bound of node 6 and 7. Calculate difference of lower and upper bound of nodes 6 and 7.

For node 6, U-L=-
32+38=6 For node
7, U-L=-38+38=0

Choose node 7, since it has minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 8, U-L=-38+38=0 For node 9, U-L=-20+20=0

Here, the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1□2□4□7□8

$X_1 = 1$

$X_2 = 1$

$X_3 = 0$

$X_4 = 1$

The solution for 0/1 knapsack problem is (x_1, x_2, x_3, x_4)=(1, 1, 0, 1) Maximum profit is:

$\sum pixi=10*1+10*1+12*0+18*1$

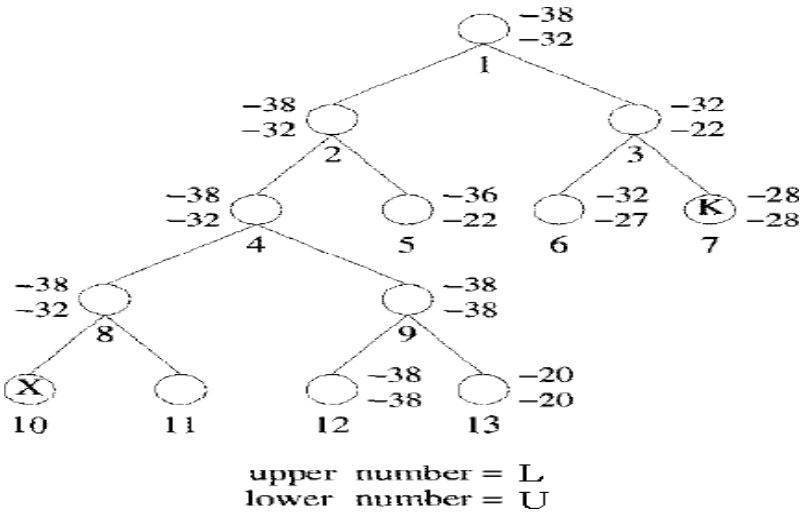
10+10+18=38.

FIFO Branch-and-Bound Solution

Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in above Example. Initially the root node, node 1 of following Figure, is the E-node and the queue of live nodes is empty. Since this is not a solution node, upper is initialized to u(l) = -32. We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of upper remains unchanged. Node 2 becomes the next E- node. Its children, nodes 4 and 5, are generated and added to the queue.

Node 3, the next-node, is expanded. Its children nodes are generated; Node 6 gets added to the queue. Node 7 is immediately killed as L (7) > upper. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then Upper is updated to u(9) = -38, Nodes 5 and 6 are the next two nodes to become B-nodes. Neither is expanded as for each, L > upper. Node 8 is the next E-node. Nodes 10 and 11 are generated; Node 10 is infeasible and so killed. Node 11 has L (11) > upper and so is also killed. Node 9 is expanded next.

When node 12 is generated, 'Upper and ans are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as L (13) > upper. The only remaining live node is node 12. It has no children and the search terminates. The value of upper and the path from node 12 to the root is output. So solution is X_1=1, X2=1, X3=0, X4=1.



TRAVELLING SALES PERSON PROBLEM

Let G = (V', E) be a directed graph defining an instance of the traveling salesperson problem. Let Cij equal the cost of edge (i, j), Cij = ∞ if (i, j) != E, and let IVI = n, without loss of generality, we can assume that every tour starts and ends at vertex 1.

Procedure for solving travelling sales person problem

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. This can be done as follows:

   Row Reduction:

   a) 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:

d) 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.

e) Find the sum of elements, which were subtracted from columns.

f) 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 α as upper bound.

2. Calculate the reduced cost matrix for every node.
   a) If path (i,j) is considered then change all entries in row **i** and column **j** of A to α.
   b) Set A(j,1) to α.
   c) Apply row reduction and column reduction except for rows and columns containing only α. Let **r** is the total amount subtracted to reduce the matrix.
   d) Find ĉ(S)=

ĉ(R)+A(i,j)+r. Repeat step 2 until all nodes are visited.

Example: Find the LC branch and bound solution for the travelling sales person problem whose cost matrix is as follows.

The cost matrix is

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Step 1: Find the reduced cost matrix

Apply now 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 t 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 =

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

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 2nd column.

The resulting column wise reduced cost matrix (A) =

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	3	12	∞

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 cost matrix. The lower bound for node is 25 and upper bound is ∞.
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), (1,5).
The tree organization up to this as follows; Variable **i** indicate the next node to visit.

Step 2:
Now consider the path (1, 2)
Change all entries of row 1 and column 2 of A to ∞ and also set A (2, 1) to ∞.

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

Apply row and column reduction for the rows and columns whose rows and column 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 ĉ(S)= ĉ(R)+A(1,2)+r □ ĉ(S)= 25 + 10 + 0 = 35.
Now consider the path (1, 3)
Change all entries of row 1 and column 3 of A to ∞ and also set A (3, 1) to ∞.

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2

15	3	∞	∞	0
11	0	∞	12	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

Row reduction sum =
0 Column reduction
sum = 11
Cumulative reduction(r) = 0
+11=11 Therefore, as ĉ(S)=
ĉ(R)+A(1,3)+r

ĉ(S)= 25 + 17 +11 = 53.

Now consider the path (1, 4)

Change all entries of row 1 and column 4 of A to ∞ and also set A(4,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

Row reduction sum =
0 Column reduction
sum = 0
Cumulative reduction(r) = 0
+0=0 Therefore, as ĉ(S)=
ĉ(R)+A(1,4)+r

ĉ(S)= 25 + 0 +0 = 25.

Now Consider the path (1, 5)

Change all entries of row 1 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	2	∞
0	3	∞	0	∞
15	3	12	∞	∞
∞	0	0	12	∞

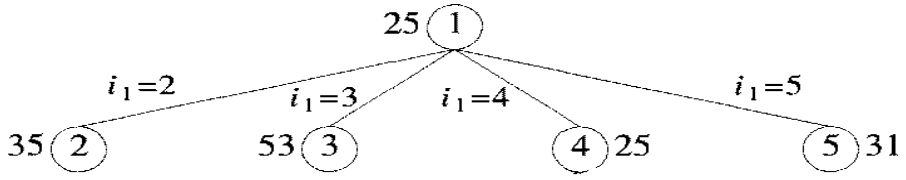
Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction(r) = 5 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(1,5)+r

ĉ(S)= 25 + 1 +5 = 31.



Numbers outside the node are $\hat{c}$ values

The tree organization up to this as follows:

The cost of the between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25, (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 =

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

The new possible paths are (4, 2), (4, 3) and (4, 5).

Now consider the path (4, 2)

Change all entries of row 4 and column 2 of A to ∞ and also set A(2,1) to ∞.

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

Row reduction sum =

0 Column reduction
sum = 0
Cumulative reduction(r) = 0
+0=0 Therefore, as ĉ(S)=
ĉ(R)+A(4,2)+r
ĉ(S)= 25 + 3 +0 = 28.

Now consider the path (4, 3)

Change all entries of row 4 and column 3 of A to ∞ and also set A(3,1) to ∞.

∞	∞	∞	∞	∞
12	∞	∞	∞	0
∞	3	∞	∞	2
∞	∞	∞	∞	∞
11	0	∞	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction(r) = 2 +11=13

Therefore, as ĉ(S)= ĉ(R)+A(4,3)+r
ĉ(S)= 25 + 12 +13 = 50.

.***Now consider the path (4, 5)***

Change all entries of row 4 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
12	∞	11	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

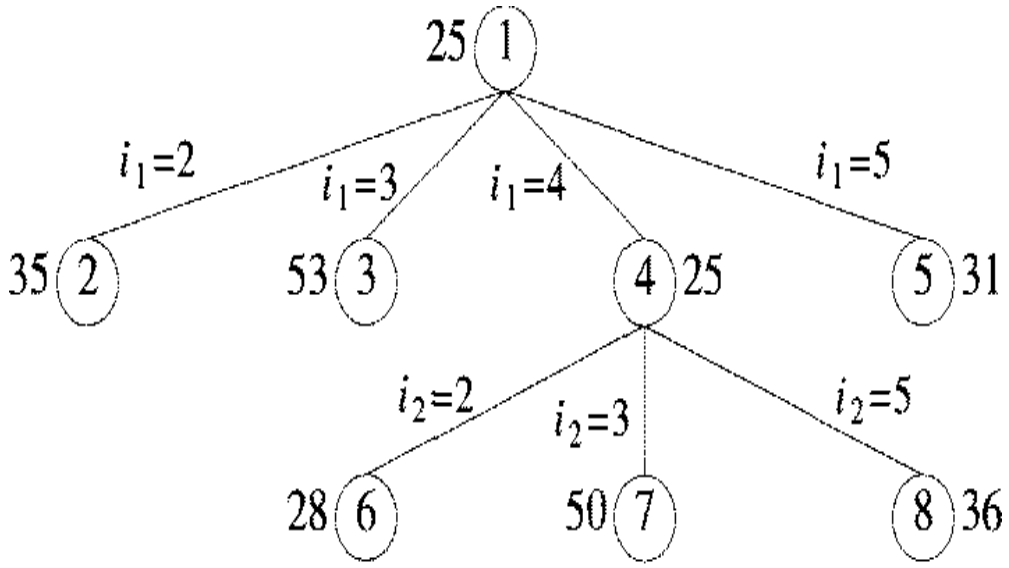
Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

is = Row reduction sum =11
Column reduction sum = 0
Cumulative reduction(r) = 11
+0=11 Therefore, as ĉ(S)=
ĉ(R)+A(4,5)+r
ĉ(S)= 25 + 0 +11 = 36.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (4, 2) = 28, (4, 3) = 50, (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.

$$A = \begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{matrix}$$

The new possible paths are (2, 3) and (2, 5).

Now Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A(3,1) to ∞.

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{matrix}$$

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{matrix}$$

Row reduction sum =13
Column reduction sum = 0
Cumulative reduction(r) = 13 +0=13
Therefore, as $\hat{c}(S)= \hat{c}(R)+A(2,3)+r$

$\hat{c}(S)= 28 + 11 +13 = 52.$

Now Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5,1) to ∞.

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

Row reduction sum =0

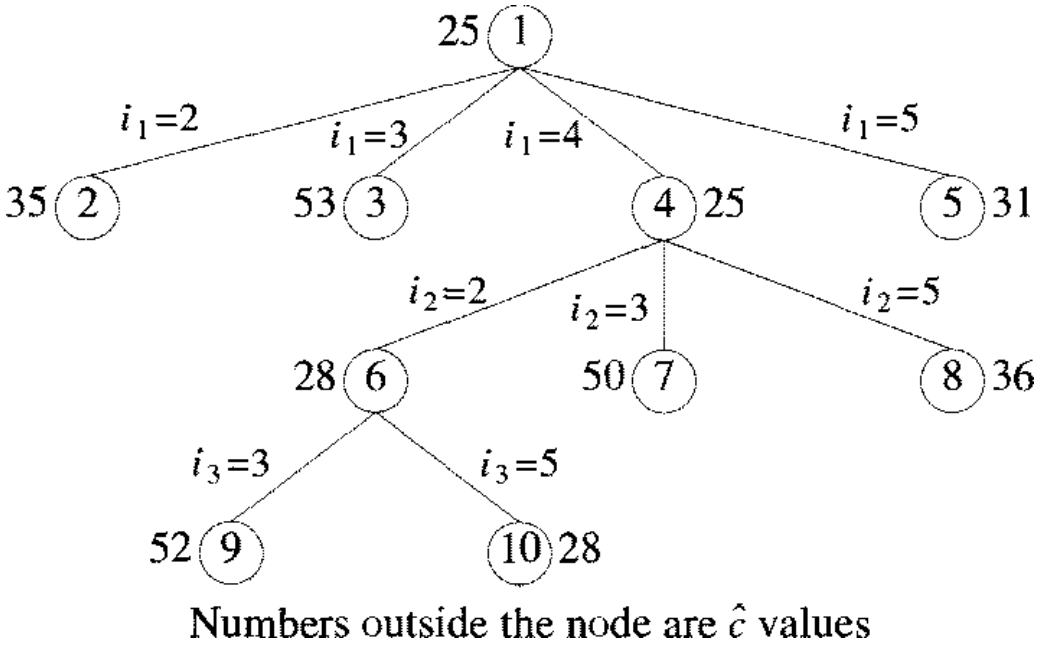
Column reduction sum

= 0

Cumulative reduction(r) = 0 +0=0

Therefore, as ĉ(S)= ĉ(R)+A(2,5)+r ĉ(S)= 28 + 0 +0 = 28.

The tree organization up to this as follows:



Numbers outside the node are $\hat{c}$ values

The cost of the between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

$$A = \begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{matrix}$$

The new possible path is (5, 3).

Now c***onsider the path (5, 3):***

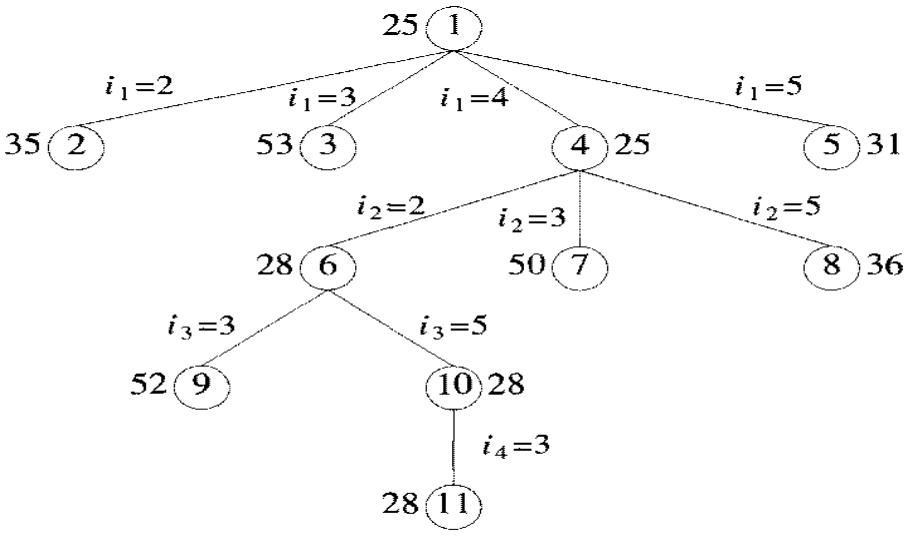
Change all entries of row 5 and column 3 of A to ∞ and also set A(3,1) to ∞. Apply row and column reduction for the rows and columns whose rows and column are not completely ∞

Then the resultant matrix is =

$$\begin{matrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{matrix}$$

Row reduction sum =0
Column reduction sum = 0
Cumulative reduction(r) = 0 +0=0
Therefore, as ĉ(S)= ĉ(R)+A(5,3)+r
ĉ(S)= 28 + 0 +0 = 28.
The path travelling sales person problem is:
1→4→2→5→3→1**:**
The minimum cost of the path is: 10+2+6+7+3=28.

The overall tree organization is as follows:



Numbers outside the node are $\hat{c}$ values

Short Answer Questions

1. Define Backtracking. Explain the state space tree with a suitable example.
2. Explain the Sum of Subsets problem and illustrate the solution space tree.
3. Describe the m-Coloring problem. Explain the constraints used in graph coloring.
4. Explain the 0/1 Knapsack problem using the Backtracking approach.
5. Define Branch and Bound. Differentiate between Live Node, Dead Node, and E-Node.
6. Compare FIFO, LIFO, and Least Cost Branch and Bound techniques.
7. Explain the concept of Bounding in Branch and Bound algorithms.
8. Describe the procedure for solving the Travelling Salesperson Problem using Least Cost Branch and Bound.

Long Answer Questions

1. Explain the Backtracking technique in detail. Discuss the construction of the state space tree and solve the **Sum of Subsets** problem with a suitable example.
2. Describe the **Graph Coloring (m-Coloring)** problem. Explain the Backtracking algorithm with a suitable example and analyze its time complexity.
3. Explain the **0/1 Knapsack Problem** using the Backtracking method with a detailed example and state space tree.
4. Discuss the **Branch and Bound** technique in detail. Explain FIFO, LIFO, and Least Cost search strategies with suitable examples.
5. Explain the **Least Cost Branch and Bound algorithm** for solving the **0/1 Knapsack Problem** with a suitable example, showing the computation of upper and lower bounds.
6. Solve the **Travelling Salesperson Problem (TSP)** using the Least Cost Branch and Bound technique. Explain the steps of row reduction, column reduction, reduced cost matrix, lower bound calculation, and optimal path determination with an example.
7. Differentiate **Backtracking** and **Branch and Bound** techniques with respect to strategy, pruning mechanism, search order, applications, advantages, and limitations. Illustrate your answer with suitable examples.
8. Explain the role of **Bounding functions** in Branch and Bound algorithms. Discuss how pruning improves efficiency in solving optimization problems such as **0/1 Knapsack** and **Travelling Salesperson Problem**.

Case Study – Unit IV

Smart Disaster Relief and Emergency Resource Allocation Using Backtracking and Branch & Bound Techniques

Course

Advanced Data Structures and Algorithm Analysis (ADS&AA)

Difficulty Level

Intermediate

Learning Outcomes

After completing this case study, students will be able to:

- Understand the concept of State Space Trees.
- Apply Backtracking techniques to constraint satisfaction problems.
- Solve optimization problems using Branch and Bound.
- Compare Backtracking and Branch & Bound approaches.
- Apply search strategies such as FIFO, LIFO, and Least Cost (LC).
- Design efficient solutions for real-world optimization problems.

Background

A **National Disaster Management Authority (NDMA)** coordinates relief operations during floods, earthquakes, and cyclones.

The authority manages:

- **20 Relief Centers**
- **150 Emergency Vehicles**
- **100 Relief Camps**
- **Thousands of Relief Packages**
- **Communication Towers**
- **Medical Supply Warehouses**

During disasters, the authority must:

- Allocate relief materials efficiently.
- Assign communication frequencies.
- Optimize transportation routes.
- Maximize the value of supplies delivered.
- Minimize transportation cost.
- Reduce response time.

Problem Statement

The NDMA requires an intelligent decision support system capable of:

- Identifying suitable combinations of relief materials.
- Assigning frequencies without interference.
- Selecting the best emergency supplies within vehicle capacity.
- Finding optimal transportation routes.
- Reducing search space using pruning techniques.
- Delivering resources with minimum operational cost.

Proposed Solution

Part A – Sum of Subsets Problem (Backtracking)

A relief vehicle has a limited carrying capacity.

Available relief kits:

Kit	Weight (kg)
Food	11
Medicine	13
Water	24
Blankets	7

The vehicle must carry exactly **31 kg** of supplies.

Backtracking explores all possible subsets and identifies only those whose total weight equals the target. For the example above, valid subsets include **(11, 13, 7)** and **(24, 7)**.

Advantages

- Finds all feasible combinations.
- Avoids duplicate solutions.
- Efficiently prunes impossible subsets.

Part B – Graph Coloring

The disaster region is divided into several communication zones.

Adjacent zones cannot use the same radio frequency because interference would disrupt rescue operations.

The communication network is modeled as a graph:

- Vertices → Communication zones
- Edges → Adjacent zones

- Colors → Radio frequencies

Using the **Graph Coloring Backtracking Algorithm**, frequencies are assigned so that neighboring regions always receive different frequencies while minimizing the number of frequencies used.

Part C – 0/1 Knapsack (Backtracking)

Each rescue vehicle has limited capacity.

Emergency supplies include:

- Oxygen cylinders
- Medicines
- Medical equipment
- Rescue kits

Each item can either be:

- Selected completely (1)
- Not selected (0)

Partial selection is not allowed.

Backtracking systematically evaluates feasible combinations to maximize the value of transported supplies.

Part D – Branch and Bound

The disaster management system must make optimal decisions quickly.

Instead of exploring every possible solution, **Branch and Bound** generates candidate solutions and eliminates (prunes) branches that cannot produce better results. This approach uses concepts such as **live nodes, E-nodes, dead nodes, and bounding functions**.

Search Strategies
1. FIFO Branch and Bound

- Uses a Queue.
- Explores nodes level by level.
- Similar to Breadth First Search.

Useful for:

- Emergency task scheduling
- Resource allocation

2. LIFO Branch and Bound

- Uses a Stack.
- Explores the most recently generated node first.
- Similar to Depth First Search.

Useful for:

- Deep exploration of possible resource allocations.

3. Least Cost (LC) Branch and Bound

Each possible transportation route is assigned a cost.

The algorithm always expands the node with the **lowest estimated cost**, using a ranking (cost) function to guide the search toward the optimal solution more quickly.

Part E – 0/1 Knapsack Using Branch and Bound

Suppose a rescue truck has:

Capacity = 15 kg

Available supplies:

Item	Weight	Benefit
Medicine	2	10
Water	4	10
Food	6	12
Oxygen Kit	9	18

Branch and Bound computes **upper and lower bounds** for partial solutions and prunes branches that cannot outperform the current best solution. This significantly reduces computation compared with exhaustive search.

Part F – Travelling Salesperson Problem (TSP)

A relief vehicle must:

- Start from the central warehouse.
- Visit every relief camp exactly once.
- Return to the warehouse.
- Minimize total travel distance.

The system models the camps as a weighted graph.

Using **Least Cost Branch and Bound**, the algorithm:

- Reduces the cost matrix.
- Computes lower bounds.
- Expands the least-cost node.
- Prunes expensive routes.

The sample solution in the unit identifies the optimal tour with **minimum cost = 28**.

Algorithm Analysis

Algorithm	Application	Typical Time Complexity
Sum of Subsets	Relief package selection	$O(2^n)$ (worst case)
Graph Coloring	Frequency assignment	$O(m^n)$ (worst case)
0/1 Knapsack (Backtracking)	Vehicle loading	$O(2^n)$
FIFO Branch & Bound	State-space exploration	Problem-dependent
LIFO Branch & Bound	Optimization search	Problem-dependent
LC Branch & Bound	Least-cost optimization	Faster in practice due to pruning
TSP (Branch & Bound)	Route optimization	Exponential in the worst case, reduced by pruning

Comparison: Backtracking vs Branch and Bound

Feature	Backtracking	Branch & Bound
Objective	Find feasible solutions	Find the optimal solution
Pruning Basis	Constraint violations	Cost bounds (upper/lower bounds)
Search Type	Depth-first	FIFO, LIFO, or Least Cost
Applications	Graph Coloring, Sum of Subsets	Knapsack, TSP, Scheduling
Optimization	Not always	Yes

Real-World Applications

These techniques are widely used in:

- Disaster Management Systems
- GPS Navigation
- Airline Route Planning
- Delivery and Logistics
- Warehouse Management
- Telecommunication Frequency Allocation
- Cloud Resource Scheduling
- Robotics Path Planning
- Military Logistics
- Smart City Infrastructure

Discussion Questions

1. Why is Backtracking suitable for solving the Sum of Subsets problem?
2. Explain how Graph Coloring helps avoid communication interference during disaster management.
3. Compare the Backtracking and Branch & Bound approaches for solving the 0/1 Knapsack problem.
4. Differentiate FIFO, LIFO, and Least Cost Branch & Bound search strategies.
5. How do upper and lower bounds reduce the search space in Branch & Bound?
6. Why is Branch & Bound preferred for optimization problems such as the Travelling Salesperson Problem?
7. Explain how pruning improves the efficiency of state-space search.
8. Suggest another real-world application where Branch & Bound can significantly reduce computational effort.

Conclusion

The **Smart Disaster Relief and Emergency Resource Allocation System** demonstrates the practical use of **Backtracking** and **Branch & Bound** techniques for solving complex optimization and constraint satisfaction problems. By applying algorithms such as **Sum of Subsets, Graph Coloring, 0/1 Knapsack, FIFO/LIFO/Least Cost Branch & Bound, and the Travelling Salesperson Problem**, organizations can efficiently allocate resources, optimize transportation, reduce operational costs, and improve emergency response times.

UNIT-V

NP Hard and NP Complete Problems: Basic Concepts, Cook‟s theorem.

NP Hard Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson Decision Problem (TSP)

NP Hard Scheduling Problems: Scheduling Identical Processors, Job Shop Scheduling

5.1 NP Hard and NP Complete Problems:

The problems are classified into two groups

Polynomial Time

1. The first group consists of the problems that can be solved in polynomial time by using deterministic algorithm

Example:

Searching of an element from an array –O(log n).

Sorting of given n elements –O(n log n)

All pairs shortest path problem-$O(n^3)$

Non-Polynomial Time

2. The second group consists of the problems that can be solved in polynomial time by using non-deterministic algorithm.

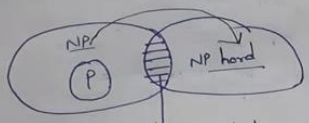
P, NP, NP Complete, NP hard problem

NP → Non-deterministic polynomial time.

P → polynomial time

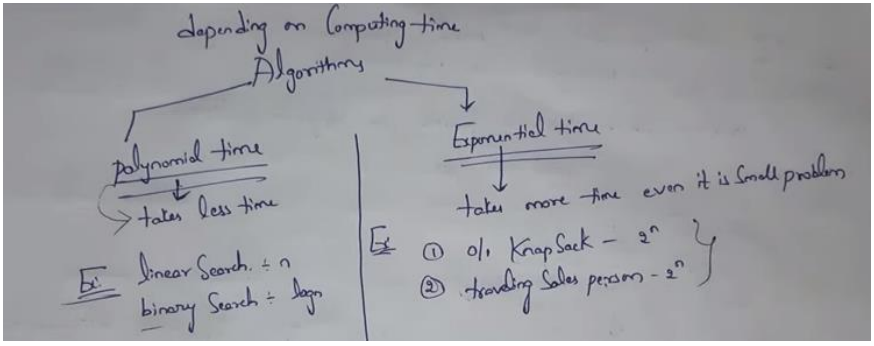
P → Problems which can be solved in polynomial time

NP → " " Cannot " " " " "



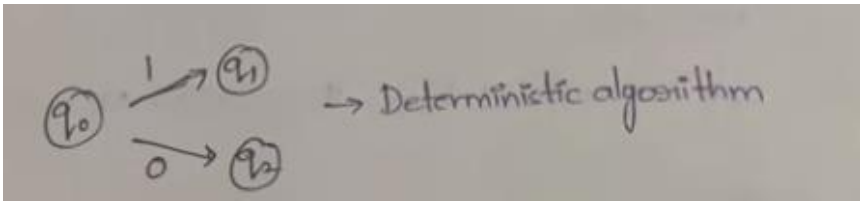
⇒ Problem which is in NP hard & NP is known as NP Complete problem

⇒ all NP problems that can be reduced to a set with in given tim
is known as NP hard problem



Deterministic Algorithm

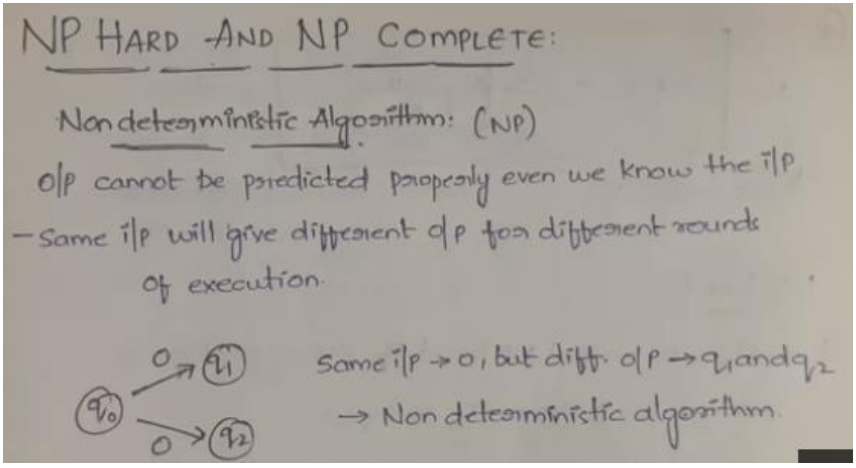
A deterministic algorithm is one where, given a particular input, the algorithm will always produce the same output and follow the same sequence of states.



Non-Deterministic Algorithm

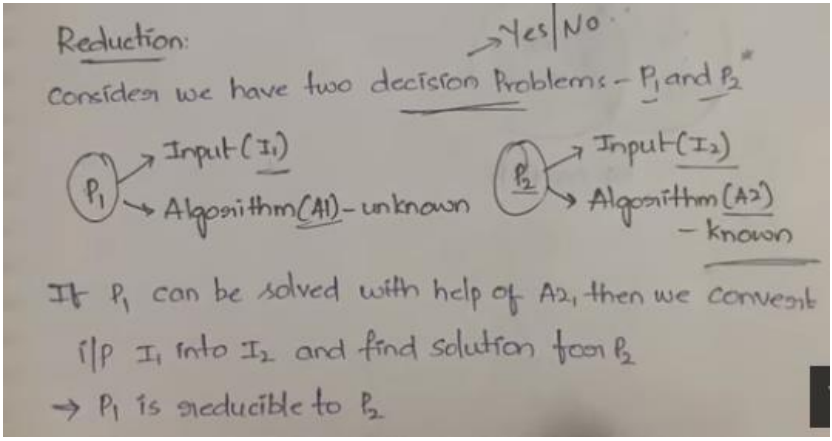
A non-deterministic algorithm is one where the same input can lead to multiple possible outcomes.

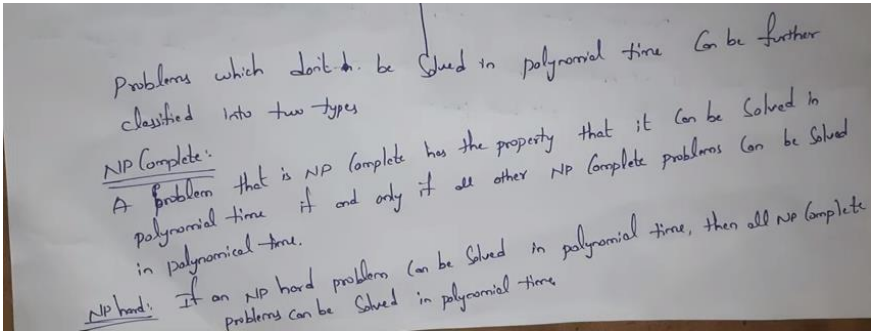
Non deterministic Algorithm (NP):



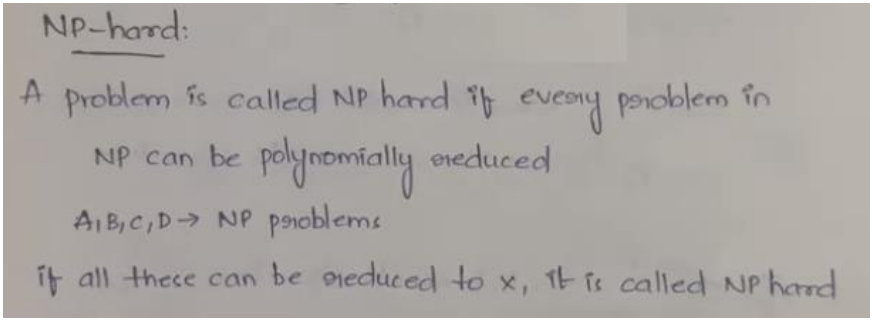
- algorithm will take more than one path.
  ∴ We cannot determine next step of execution.
- only approximate solutions, no accurate solutions.
  Ex: Monte Carlo Problem, Genetic Algorithm etc

Reduction or Reducible

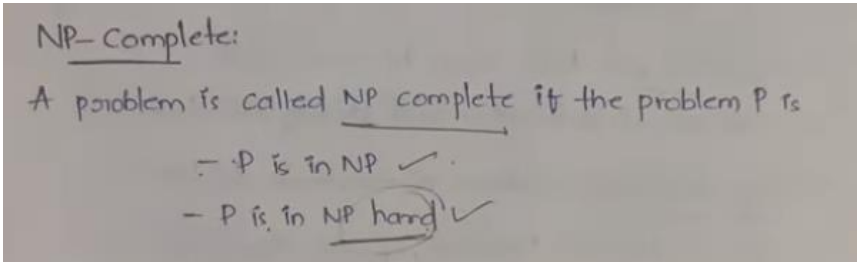


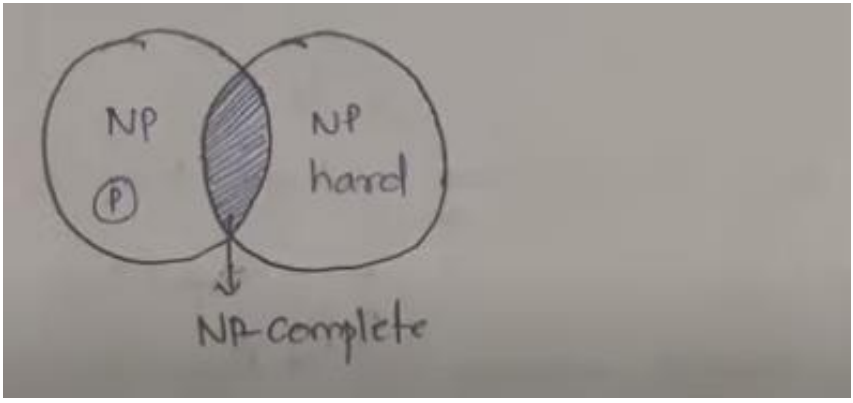


NP-Hard



NP-Complete





P-Class problems (Deterministic algorithms)

Problems that can be solved in polynomial time are called P-class problems. Where „p" stand for polynomial time.

Example:

1. Searching of key element among n number of elements- O(n)
2. Sorting of „n" elements- $O(n^2)$
3. Addition of two matrices- $O(n^2)$
4. Multiplication of matrices- $O(n^3)$
5. All pairs shortest path problem- $O(n^3)$

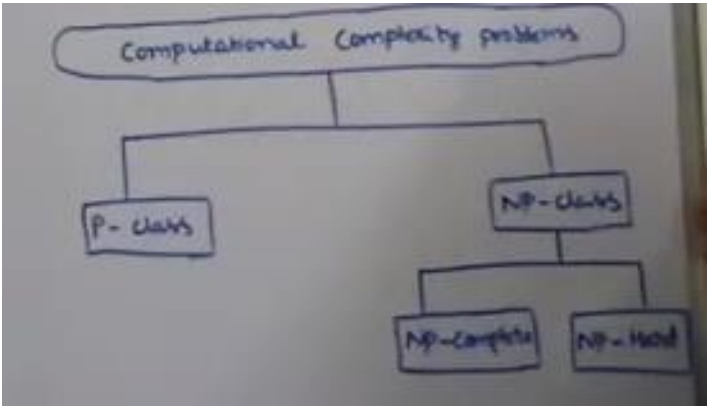
These problems can be solved by using deterministic algorithms.

NP-Class Problems (Non deterministic algorithm)

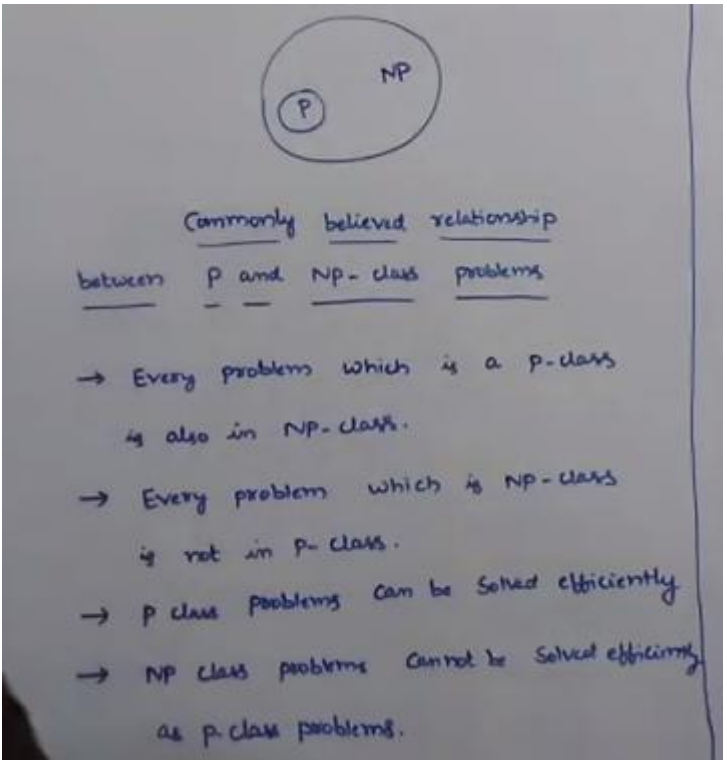
Problems that can be solved in polynomial time by using Non-deterministic algorithms are called NP-class problems. Where „NP" stand for Non-Polynomial time.

Example:

1. Travelling salesperson problem
2. 0/1 knapsack problem
3. Graph coloring problem
4. Hamiltonian cycle problem



Relationship between P and NP-class problem



5.2 Cook’s theorem.

Cook's theorem states that the problems of satisfiability and determining if P=NP are equivalent. It proves this by showing that if satisfiability is in P (can be solved in polynomial time), then any problem in NP (can be solved by a non-deterministic Turing machine in polynomial time) can also be solved in P, meaning P=NP. Conversely, if P=NP, then satisfiability must also be in P.

Statement

The satisfiability problem (SAT) is NP complete.

In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

Boolean Satisfiability Problem

Boolean Satisfiability or simply SAT is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

Satisfiable: If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.

Unsatisfiable : If it is not possible to assign such values, then we say that the formula is unsatisfiable.

Examples:

- $F = A \wedge \bar{B}$, is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
- $G = A \wedge \bar{A}$, is unsatisfiable, because:

A	$\bar{A}$	G
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

Note: Boolean satisfiability problem is NP-complete

Cooks Theorem

- Stephen Cook introduced the notion of NP-Complete Problems.
  - This makes the problem "P = NP ?" much more interesting to study.
  - L is **NP-Complete** *if $L \in NP$ then for all other $L' \in NP, L' \propto L$*
  - L is **NP-Hard** if *for all other $L' \in NP, L' \propto L$*
  - If an NP-complete problem can be solved in polynomial time then all problems in NP can be solved in polynomial time.
  - If a problem in NP cannot be solved in polynomial time then all problems in NP-complete cannot be solved in polynomial time.
  - Note that an NP-complete problem is one of those hardest problems in NP.
- Lemma :
  If L1 and L2 belong to NP,
  L1 is NP-complete, and L1 $\propto$ L2
  then L2 is NP-complete.
  i.e. $L1, L2 \in NP$ and for all other $L' \in NP, L' \propto L1$ and $L1 \propto L2 \rightarrow L' \propto L2$

Cooks Theorem and Satisfiability

- SATISFIABILITY is NP-Complete. (The first NP-Complete problem)
  - Instance : Given a set of variables, U, and a collection of clauses, C, over U.
  - Question : Is there a truth assignment for U that satisfies all clauses in C?
- CNF-Satisfiability (CNF – Conjunctive Normal Form)
  - because the expression is in (the product of sums).
- Example:

$$\text{“}\neg x_i\text{”} = \text{“not } x_i\text{”} \quad \text{“OR”} = \text{“logical or”} \quad \text{“AND”} = \text{“logical and”} \quad U = \{x_1, x_2\}$$

$$C_1 = \{(x_1, \neg x_2), (\neg x_1, x_2)\}$$
$$= (x_1 \text{ OR } \neg x_2) \text{ AND } (\neg x_1 \text{ OR } x_2)$$
$$\text{if } x_1 = x_2 = \text{True} \rightarrow C_1 = \text{True}$$

$$C_2 = (x_1, x_2)\ (x_1, \neg x_2)\ (\neg x_1) \rightarrow \text{not satisfiable}$$

5.3 NP Hard Graph Problems:

5.3.1 Clique Decision Problem (CDP)

5.3.2 Chromatic Number Decision Problem (CNDP)

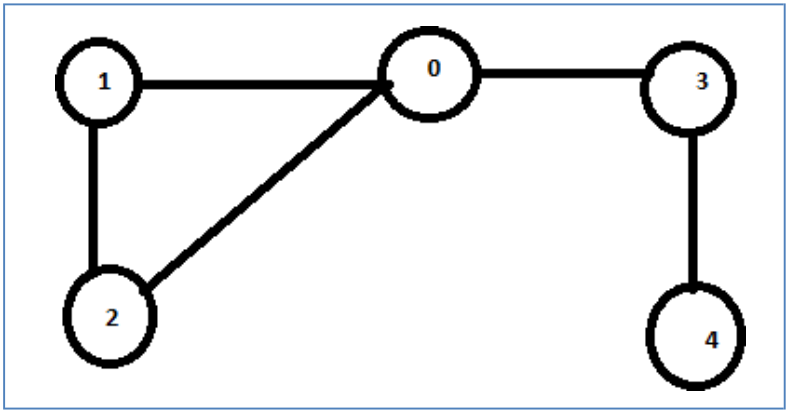
5.3.3 Traveling Salesperson Decision Problem (TSP).

5.3.1 Clique Decision Problem (CDP)

A clique is a sub-graph of graph such that all vertices in sub-graph are completely connected with each other.

The max-clique problem is the computational problem of finding maximum clique of the graph.

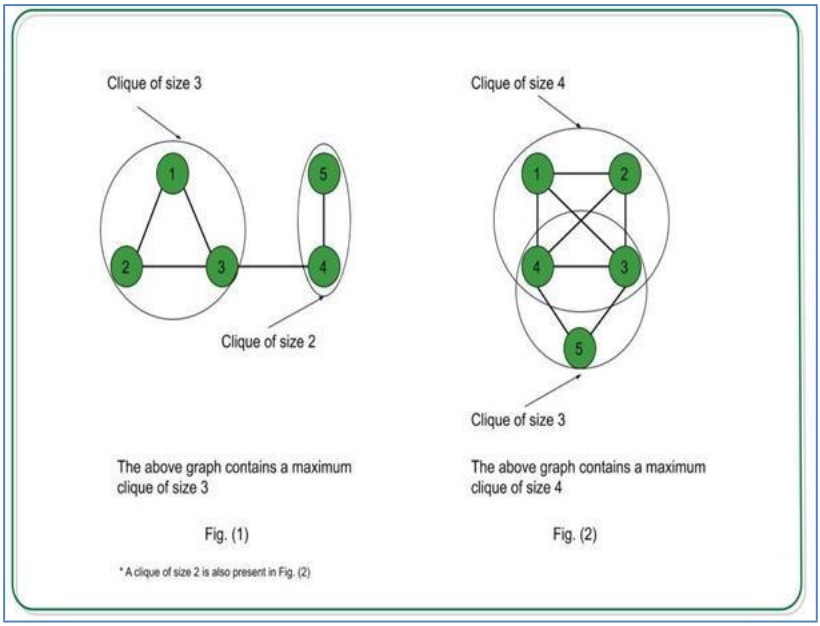
Example



This graph can be divided into two cliques

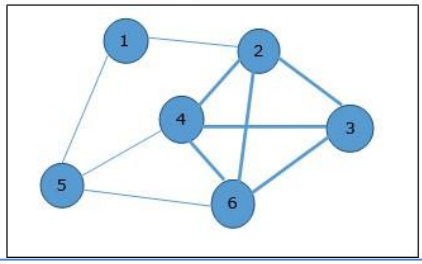
One clique contains {0,1,2}

Other clique contains{3,4}



Example

Take a look at the following graph. Here, the sub-graph containing vertices 2, 3, 4 and 6 forms a complete graph. Hence, this sub-graph is a **clique**. As this is the maximum complete sub-graph of the provided graph, it's a **4-Clique**.



5.3.2 Chromatic Number Decision Problem (CNDP)

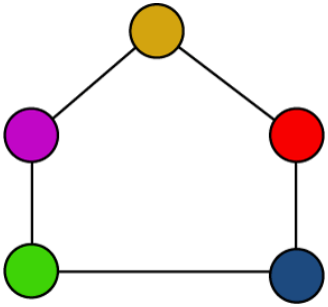
Graph Coloring Algorithm

- There exists no efficient algorithm for coloring a graph with minimum number of colors.
- Graph Coloring is a NP complete problem.
- However, a following greedy algorithm is known for finding the chromatic number of any given graph.
- Graph coloring can be described as a process of assigning colors to the vertices of a graph.
- In this, the same color should not be used to fill the two adjacent vertices.
- We can also call graph coloring as Vertex Coloring.
- In graph coloring, we have to take care that a graph must not contain any edge whose end vertices are colored by the same color.
- This type of graph is known as the Properly colored graph.

Graph coloring can be described as a process of assigning colors to the vertices of a graph. In this, the same color should not be used to fill the two adjacent vertices. We can also call graph coloring as Vertex Coloring. In graph coloring, we have to take care that a graph must not contain any edge whose end vertices are colored by the same color. This type of graph is known as the properly colored graph.

Example of Graph coloring

In this graph, we are showing the properly colored graph, which is described as follows:



The above graph contains some points, which are described as follows:

- The same color cannot be used to color the two adjacent vertices.
- Hence, we can call it as a properly colored graph.

Applications of Graph coloring

There are various applications of graph coloring. Some of their important applications are described as follows:

- Assignment
- Map coloring
- Scheduling the tasks
- Sudoku
- Prepare time table
- Conflict resolution

Chromatic Number decision problem

Chromatic Number

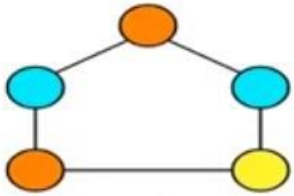
- Chromatic Number is the minimum number of colors required to properly color any graph.

OR

- Chromatic Number is the minimum number of colors required to color any graph such that no two adjacent vertices of it are assigned the same color.

Chromatic Number Example-

- Consider the following graph-



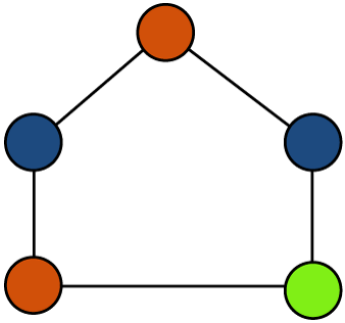
Chromatic Number = 3

In this graph,

- No two adjacent vertices are colored with the same color.
- Minimum number of colors required to properly color the vertices = 3.
- Therefore, Chromatic number of this graph = 3.
- We can not properly color this graph with less than 3 colors.

Example of Chromatic number:

To understand the chromatic number, we will consider a graph, which is described as follows:



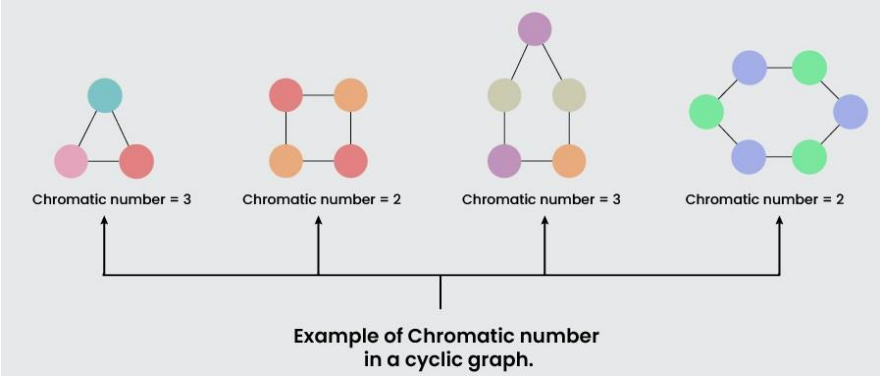
The above graph contains some points, which are described as follows:

The same color is not used to color the two adjacent vertices.

The minimum number of colors of this graph is 3, which is needed to properly color the vertices.

Hence, in this graph, the chromatic number = 3

If we want to properly color this graph, in this case, we are required at least 3 colors.



Example of Chromatic number in a cyclic graph.

5.3.3 Traveling Salesperson Decision Problem (TSP).

The NP-hard Traveling Salesman Problem (TSP) asks to find the shortest route that visits all vertices in the graph. To be precise, the TSP is the shortest tour that visits all vertices and returns back to the start.

5.4 NP Hard Scheduling Problems:

5.4.1 Scheduling Identical Processors

5.4.2 Job Shop Scheduling

5.4.1 Scheduling Identical Processors

5.4.2 Job Shop Scheduling

Job Shop Scheduling Problem (JSSP), which aims to schedule several jobs over some machines in which each job has a unique machine route, is one of the NP-hard optimization problems researched over decades for finding optimal sequences over machines.

https://optimization.cbe.cornell.edu/index.php?title=Job_shop_scheduling

Short Answer Questions

1. Define P-Class and NP-Class problems. Differentiate between deterministic and non-deterministic algorithms with suitable examples.
2. Explain the concepts of NP-Hard and NP-Complete problems. How are they related?
3. State Cook's Theorem and explain its significance in computational complexity theory.
4. Describe the Boolean Satisfiability (SAT) problem. Differentiate between satisfiable and unsatisfiable Boolean formulas with examples.
5. Define the Clique Decision Problem (CDP). Illustrate the concept of a clique using a suitable graph.
6. Explain the Chromatic Number Decision Problem (CNDP) and discuss its practical applications.
7. Describe the Traveling Salesperson Decision Problem (TSP). Why is it considered an NP-Hard problem?
8. Explain Scheduling Identical Processors and Job Shop Scheduling with suitable real-world examples.

Long Answer Questions

1. Explain the classification of computational problems into Polynomial-Time and Non-Polynomial-Time problems. Compare deterministic and non-deterministic algorithms with suitable examples.
2. Discuss the concepts of P-Class, NP-Class, NP-Hard, and NP-Complete problems. Illustrate their relationships using a suitable diagram.

3. Explain the concept of reduction in computational complexity. Why is polynomial-time reduction important in proving NP-Completeness?
4. State and explain Cook's Theorem. Discuss why the Boolean Satisfiability (SAT) problem is considered the first NP-Complete problem.
5. Explain the Boolean Satisfiability Problem with suitable examples. How does Cook's Theorem establish the NP-Completeness of SAT?
6. Explain the Clique Decision Problem (CDP) with an appropriate graph. Discuss its computational complexity and applications.
7. Describe the Chromatic Number Decision Problem (CNDP). Explain graph coloring with a suitable example and discuss its applications in scheduling and map coloring.
8. Explain the Traveling Salesperson Decision Problem (TSP). Discuss why it is computationally difficult and describe its real-world applications.
9. Explain the Scheduling Identical Processors problem. Discuss the challenges involved in obtaining an optimal schedule and its practical applications.
10. Describe the Job Shop Scheduling Problem (JSSP). Explain why it is NP-Hard and discuss its importance in manufacturing and production industries.

Case Study – Unit V

Smart Manufacturing and Cloud Computing Resource Optimization Using NP-Hard and NP-Complete Problems

Course

Advanced Data Structures and Algorithm Analysis (ADS&AA)

Difficulty Level

Intermediate

Learning Outcomes

After completing this case study, students will be able to:

- Differentiate between P, NP, NP-Hard, and NP-Complete problems.
- Understand Cook's Theorem and polynomial-time reductions.
- Apply Clique and Graph Coloring concepts to real-world applications.
- Analyze the Traveling Salesperson Decision Problem.
- Solve processor scheduling and job shop scheduling problems.
- Evaluate optimization techniques for complex computational problems.

Background

TechSmart Industries is a multinational company that operates:

- **5 Manufacturing Plants**
- **20 Production Lines**

- **500 Machines**
- **300 Industrial Robots**
- **50 Cloud Servers**
- **Thousands of Customer Orders per Day**

The company wants to automate production planning, machine scheduling, transportation, and cloud resource allocation.

However, many of these optimization problems become extremely difficult as the number of jobs and resources increases.

Problem Statement

The company requires a decision support system that can:

- Classify computational problems based on complexity.
- Allocate cloud resources efficiently.
- Schedule manufacturing jobs.
- Optimize delivery routes.
- Allocate communication channels.
- Detect highly connected groups in industrial networks.
- Minimize production time and operational costs.

Proposed Solution

Part A – Classification of Computational Problems

The company classifies its computational tasks into four categories.

P-Class Problems

Problems that can be solved efficiently using deterministic algorithms.

Examples within the company include:

- Searching employee records
- Sorting inventory
- Matrix operations
- Shortest path calculations

These problems have polynomial-time solutions and are considered computationally tractable.

NP-Class Problems

Some planning tasks become significantly more complex.

Examples include:

- Delivery route planning
- Machine scheduling
- Graph coloring
- Knapsack optimization

Although verifying a proposed solution is efficient, finding the optimal solution is computationally challenging.

NP-Hard Problems

Several optimization problems in manufacturing and logistics belong to the NP-Hard class.

Examples include:

- Job Shop Scheduling

- Traveling Salesperson Problem
- Graph Coloring

These problems generally have no known polynomial-time algorithms and often require approximation or heuristic methods.

NP-Complete Problems

Some decision problems are both:

- In NP
- NP-Hard

These are known as NP-Complete problems and are among the most challenging computational decision problems.

Part B – Cook's Theorem

The company's research team studies whether a new optimization problem belongs to the NP-Complete class.

Using **Cook's Theorem**, they reduce the new problem to the Boolean Satisfiability (SAT) problem.

If the reduction can be completed in polynomial time and SAT is solvable, then the new problem is also considered NP-Complete. Cook's Theorem establishes that **SAT is the first NP-Complete problem**.

Part C – Clique Decision Problem (CDP)

The company maintains a communication network among production machines.

Each machine is represented as:

- Vertex → Machine
- Edge → Direct communication

Management wants to identify whether there exists a group of **k machines** that are all directly connected.

This is modeled using the **Clique Decision Problem**, where a clique represents a fully connected subgraph. Such analysis helps identify tightly collaborating machine clusters.

Part D – Chromatic Number Decision Problem (CNDP)

Different manufacturing robots share wireless communication channels.

Adjacent robots must use different frequencies to avoid interference.

The communication network is represented as a graph.

The objective is to determine whether all robots can be assigned frequencies using at most **k colors (frequencies)**.

This is modeled using the **Chromatic Number Decision Problem**. Typical applications include:

- Frequency assignment
- Examination timetabling
- Task scheduling

- Map coloring

Part E – Traveling Salesperson Decision Problem (TSP)

Every day, a delivery vehicle must:

- Start from the warehouse.
- Visit all customer locations.
- Return to the warehouse.
- Ensure the total travel distance does not exceed a specified limit.

Instead of asking for the shortest route, the **decision version** asks:

"Is there a tour with total cost less than or equal to K?"

This decision formulation is widely used in logistics and transportation planning.

Part F – Scheduling Identical Processors

The company owns **20 identical processors** for cloud-based production monitoring.

Hundreds of computational tasks arrive every minute.

The scheduler must:

- Assign tasks to processors.
- Balance workloads.
- Minimize completion time.
- Avoid processor overload.

This scheduling problem is NP-Hard because finding the optimal assignment becomes computationally expensive as the number of tasks increases.

Part G – Job Shop Scheduling

The factory manufactures:

- Automobiles
- Electronic Devices
- Industrial Equipment

Each product must pass through multiple machines in a specific order.

Example:

Job	Machine Sequence
J1	M1 → M3 → M5
J2	M2 → M4 → M1
J3	M5 → M2 → M3

The objective is to:

- Minimize total production time (Makespan).
- Reduce machine idle time.
- Increase factory productivity.

Because every job has its own machine sequence and resource constraints, the **Job Shop Scheduling Problem (JSSP)** is NP-Hard.

Complexity Classification

Problem	Complexity Class	Real-World Application
Searching	P	Employee database
Sorting	P	Inventory management
Matrix Multiplication	P	Scientific computing
SAT	NP-Complete	Logic verification
Clique Decision	NP-Complete	Social & communication networks
Chromatic Number Decision	NP-Complete	Frequency allocation
TSP Decision	NP-Complete	Logistics planning
Processor Scheduling	NP-Hard	Cloud computing
Job Shop Scheduling	NP-Hard	Manufacturing systems

Real-World Applications

These concepts are used in:

- Google Cloud
- Microsoft Azure
- Amazon Web Services (AWS)
- Manufacturing Automation
- Airline Scheduling
- Hospital Resource Management
- Smart Grid Systems
- Mobile Network Frequency Allocation
- Examination Timetabling
- Supply Chain and Logistics

Discussion Questions

1. Differentiate between P-Class and NP-Class problems with suitable examples.
2. Explain the significance of Cook's Theorem in computational complexity.

3. How does the Clique Decision Problem help identify fully connected groups in industrial communication networks?
4. Why is the Chromatic Number Decision Problem important in wireless communication systems?
5. Explain the Traveling Salesperson Decision Problem and its role in logistics optimization.
6. Why are Scheduling Identical Processors and Job Shop Scheduling considered NP-Hard problems?
7. Compare NP-Hard and NP-Complete problems with suitable examples.
8. Suggest suitable heuristic or approximation approaches that industries can use when exact solutions to NP-Hard problems are computationally infeasible.

Conclusion

The **Smart Manufacturing and Cloud Computing Resource Optimization System** demonstrates how computational complexity theory guides the design of efficient algorithms for large-scale industrial applications. Concepts such as **P-Class, NP-Class, NP-Hard, NP-Complete Problems, Cook's Theorem, Clique Decision Problem, Chromatic Number Decision Problem, Traveling Salesperson Decision Problem, Scheduling Identical Processors, and Job Shop Scheduling** help engineers understand which problems can be solved efficiently and which require approximation, heuristics, or advanced optimization techniques. These concepts form the foundation of modern scheduling, cloud computing, logistics, manufacturing, and network optimization systems.