UNIT 1

TWO MARKS QUESTIONS

1. *What is an Algorithm? What are the criteria for writing an algorithms?*

:-"An algorithm is a sequence of unambiguous instructions for solving the problem and to obtain a required output for any legitimate input in finite amount of time".

• An algorithm can take zero or more inputs.
• It should give one or more outputs.
• Definiteness: Each instruction should be clear and unambiguous.
• Finiteness: All cases the algorithm must terminate in finite number of steps.
• Effectiveness: It must be simple and feasible.

2. *What are the methods of specifying an algorithms?*

:-There are three options that are most widely used for specifying algorithms.
– Using Natural Language
– Using Pseudo codes
– Using Flow Charts

3. *List the steps of Algorithm design and analysis process*

:- – Understand the problem
– Ascertaining the capabilities of a Computational device
– Choosing between the Exact and Approximate Problem solving
– Deciding appropriate data structure
– Algorithm design techniques
– Methods of Specifying an Algorithm
– Proving algorithms correctness
– Analyzing an Algorithm
– Coding an algorithm

4. *What is exact algorithm and approximation algorithm? Give example*.

:-The algorithms which solve the problem exactly are called exact algorithms.
• The algorithms which solve the problem approximately are called approximation algorithms.

5. List the important Problem Types.

:-1. Sorting
2. Searching
3. String Processing
4. Graph Problems
5. Combinational Problems
6. Geometric Problems
7. Numerical Problems

6. *Define the different methods for measuring algorithm efficiency.*

:-Efficiency depends on two factors:
Time efficiency
Space efficiency

7. *Write the Euclid algorithm to find the GCD of 2 numbers.*

:-//Algorithm: Euclid(m,n) an algorithm to find gcd of 2 numbers
// Two positive numbers m and n are the input.
// Output: greatest common divisor of m and n.
Step1. If n=0 return the value of m as the answer and stop else proceed to step 2.
Step2. Divide m by n and assign the value of the remainder to r.
Step3. Assign the value of n to m and the value of r to n. Go to step 1.

8. *What are combinatorial problems? Give an example.*

:-The travelling sales men problem and the graph controlling problem are examples.
• These problems ask to find a combinatorial object such as a permutation, a combination or a subset – that satisfies certain constraints and has some desired property.

9. *What are following data structures?*

a) Single linked list:-• A special node called header is used to start a linked list.
• Each node contains two parts, one for data and another for address of next node.
• Last node contains null.
b) double linked list :-Each node contains left and right pointer, except the first and last, which contains null as shown below.
c) stack :-A stack is a list in which insertion and deletion can be done only from

one side called top of stack. (LIFO)
d) queue:-A queue is a list in which insertion is done from one end called the rear and deletion is done from the other end called front. (FIFO)
e) graph :-A graph G=(V,E) is defined by a pair of two sets: a finite set V of items called vertices and asset E of pairs of these items called edges.
f) tree:-It's a connected acyclic graph.

10.Explain the terms (w.r.t graph):
a) Directedgraph:-If the pair of vertices are ordered then the graph is called directed.
• Example: Vertices (u,v) is directed from u to v
• Directed graphs also called as Digraphs.
b) undirected graph:-If the pair of vertices is unordered then the graph is undirected.
• Example: vertices (u,v) is as same as (v,u)).
• E={(a,b),(a,c),(b,f),(b,d),(d,e),(f,e),(f,c)}
• V={a,b,c,d,e,f}
c) adjacency matrix:-The adjacency matrix of a graph with n vertices is an n by n Boolean matrix with one row and column for each of the graph's vertices in which the element in the ith row and jth column is equal to 1 if there is an edge from
the ith vertex to jth vertex and equal to zero if there is no edge.
d) adjacency lists :-It's a collection of linked lists one for each vertex,that contain all the vertices adjacent to the list's vertex.
• Usually such lists starts with a header identifying a vertex for which the list is compiled.
e) weighted graph :-It's a graph with numbers assigned to its edges.
• These numbers are called as weights. With this many
real life applications can be represented.
• Example:
– Shortest path between two paths in transportation network.
f) path :-A path from vertex u to vertex v of a graph can be defined as a sequence of adjacent vertices that starts with u and ends with v.
• If all the edges of a path are distinct, the path is called simple path.
g) cycle:-It is a simple path of a positive length that starts and ends at the same vertex.
• A graph with no cycle is said to be acyclic.

11. Explain the terms (w.r.t trees)

*a) free tree :-*It's a tree which has no roots.

*b) forest :-*It's a graph that has no cycles but not necessarily connected.

c) rooted tree :-In a tree it is possible to select an arbitrary vertex and consider is as the root. Such tree is called as rooted tree.

d) ordered tree :-An ordered tree is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right.

*e) binary search tree:-*If is larger than all the numbers in its left subtreeand smaller than all the numbers in its right subtree then such trees are called binary search trees.

12.Define Sets and Dictionaries.

A set can be described as an unordered collection (possibly empty) of distinct items called elements of the set.

• Ex: S = {2,3,5,7})

• A Data structure that implements – searching of a given item, adding a new item, and deleting an item,is called as Dictionary.

13. *Define the two types of efficiencies used in algorithm.*

:-Time effeciency,space efficiency

14.What are Best case and Worst case in algorithm?

:-The best-case efficiency of an algorithm for the input size n for which the algorithm takes least time or the fastest among all possible inputs of that size.

– Cbest (n) = 1

The worst-case efficiency of an algorithm is its efficiency for input of size n, for which the algorithm runs the longest among all possible inputs of that size.

– Cworst (n) = n

15. *Why order growth necessary in algorithm analysis?*

:-The table helps to compare among all the functions.

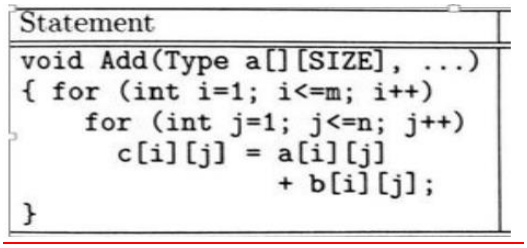
• The function growing the slowest among these is the logarithmic function. (log2n)

• It grows so slowly, in fact, that we should expect a program implementing an algorithm with a logarithmic basic-operation count to run instantaneously on inputs

of all realistic sizes.

16. *What are asymptotic notation? Why it is required?*
:-Asymptotic notations are used to compare the orders of growth.
• Three notations:
• O(big oh)
• Ω(big omega)
• Θ(big theta)

17.Find the time complexity for given algorithms



Statement

void Add(Type a[][SIZE], ...)
{ for (int i=1; i<=m; i++)
for (int j=1; j<=n; j++)
c[i][j] = a[i][j]
+ b[i][j];
}

18.What is Big O notation? Give an example
:-O(g(n)) is the set of all functions with a smaller or same order of growth as g(n).
Example: n Є 0(n2), n(n-1)/2 Є 0(n2), n3 Є/ 0(n2)

19.What is Big Omega notation? Give an example
:-Ω(g(n)) stands for the set of all functions with a larger or same order of growth as g(n).
Example: n3 Є Ω (n2) , n(n-1)/2 Є Ω(n2), 100n+5 Є/ Ω(n2)

20.Define Big Theta notation. Give an example
:-Θ(g(n)) is the set of all functions that have the sameorder of growth as g(n)
Example: Every quadratic function an
2+bn+c with a>0 is in(n2)

21.Define Little Oh notation. Give an example
:-Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).
Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer

constant n0 ≤ 1 such that f(n) > 0.

22.What is recurrence relation? Give an example.

:-recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.
For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

23.Prove the following statements.

a) 100n + 5 =`O(n2)

To prove that 100n + 5 is not in O(n^2), we can use the formal definition of big O notation.
The statement "100n + 5 is in O(n^2)" means that there exist constants C and k such that for all values of n greater than or equal to k, the following inequality holds:
|100n + 5| <= C * |n^2|
Now, let's try to find C and k that make this inequality true.
|100n + 5| <= C * |n^2|
We can simplify the left side of the inequality by taking the absolute value:
100n + 5 <= C * |n^2|
We can ignore the constant factor of 5, as it won't affect the growth rate:
100n <= C * |n^2|
Divide both sides by n:
100 <= C * |n|
Now, let's see if we can find values of C and k that satisfy this inequality. The problem here is that for any positive value of C and any value of n, the right side of the inequality C * |n| will always be greater than or equal to 100. Therefore, there is no constant C and value of k that will make this inequality true for all n.
So, 100n + 5 is not in O(n^2) because there is no constant C and value of k that satisfies the formal definition of big O notation for this function.

b*) n2 + 5n + 7= Θ(n2)*

To prove that n^2 + 5n + 7 is in Θ(n^2), we need to show that it is both an upper bound and a lower bound for n^2.
1. Upper Bound (O-notation):

We can prove that n^2 + 5n + 7 is O(n^2) if we can find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:
n^2 + 5n + 7 <= C * n^2
Now, let's simplify the inequality:
n^2 + 5n + 7 <= C * n^2
Divide both sides by n^2:
1 + 5/n + 7/n^2 <= C
As n approaches infinity, the terms 5/n and 7/n^2 go to zero. So, we have:
1 <= C
This is true for any positive constant C, which means n^2 + 5n + 7 is indeed O(n^2).
2. Lower Bound (Ω-notation):
We also need to prove that n^2 + 5n + 7 is Ω(n^2). This means we need to find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:
n^2 + 5n + 7 >= C * n^2
Now, let's simplify this inequality:
n^2 + 5n + 7 >= C * n^2
Divide both sides by n^2:
1 + 5/n + 7/n^2 >= C
As n approaches infinity, the terms 5/n and 7/n^2 go to zero. So, we have:
1 >= C
This is also true for any positive constant C, which means n^2 + 5n + 7 is Ω(n^2).
Since n^2 + 5n + 7 is both O(n^2) and Ω(n^2), it is indeed Θ(n^2).

c) n2 + n =O(n3)

To prove that n^2 + n is in O(n^3), we need to find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:
n^2 + n <= C * n^3
Now, let's simplify the inequality:
n^2 + n <= C * n^3
Divide both sides by n^3:
1/n + 1 <= C
As n approaches infinity, the term 1/n goes to zero, and we have:
1 <= C
This is true for any positive constant C, which means n^2 + n is indeed O(n^3).
So, n^2 + n is in O(n^3) with constants C = 1 and any value of k.

d) ½ n(n-1)=Θ(n2)

To prove that ½ n(n-1) is in Θ(n^2), we need to show that it is both an upper bound and a lower bound for n^2.

1. Upper Bound (O-notation):

We can prove that ½ n(n-1) is O(n^2) if we can find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:

½ n(n-1) <= C * n^2

Now, let's simplify the inequality:

½ n(n-1) <= C * n^2

Divide both sides by n^2:

½ (1 - 1/n) <= C

As n approaches infinity, the term 1/n goes to zero, and we have:

½ <= C

This is true for any positive constant C, which means ½ n(n-1) is indeed O(n^2).

2. Lower Bound (Ω-notation):

We also need to prove that ½ n(n-1) is Ω(n^2). This means we need to find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:

½ n(n-1) >= C * n^2

Now, let's simplify this inequality:

½ n(n-1) >= C * n^2

Divide both sides by n^2:

½ (1 - 1/n) >= C

As n approaches infinity, the term 1/n goes to zero, and we have:

½ >= C

This is also true for any positive constant C, which means ½ n(n-1) is Ω(n^2).

Since ½ n(n-1) is both O(n^2) and Ω(n^2), it is indeed Θ(n^2).

e) 5n2 + 3n + 20 =O(n2)

To prove that 5n^2 + 3n + 20 is in O(n^2), we need to find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:

5n^2 + 3n + 20 <= C * n^2

Now, let's simplify the inequality:

5n^2 + 3n + 20 <= C * n^2

Subtract C * n^2 from both sides:

3n + 20 <= (C - 5) * n^2
Divide both sides by n^2:
(3/n) + (20/n^2) <= C - 5
As n approaches infinity, both (3/n) and (20/n^2) go to zero. So, we have:
0 <= C - 5
For this inequality to be true for all n, C should be greater than or equal to 5.
Therefore, 5n^2 + 3n + 20 is in O(n^2) with constants C >= 5 and any value of k.

f) ½n2+3n=Θ(n2)

To prove that ½n^2 + 3n is in Θ(n^2), we need to show that it is both an upper bound and a lower bound for n^2.
1. Upper Bound (O-notation):
We can prove that ½n^2 + 3n is O(n^2) if we can find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:
½n^2 + 3n <= C * n^2
Now, let's simplify the inequality:
½n^2 + 3n <= C * n^2
Divide both sides by n^2:
½ + 3/n <= C
As n approaches infinity, the term 3/n goes to zero, and we have:
½ <= C
This is true for any positive constant C, which means ½n^2 + 3n is indeed O(n^2).
2. Lower Bound (Ω-notation):
We also need to prove that ½n^2 + 3n is Ω(n^2). This means we need to find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:
½n^2 + 3n >= C * n^2
Now, let's simplify this inequality:
½n^2 + 3n >= C * n^2
Divide both sides by n^2:
½ + 3/n >= C
As n approaches infinity, the term 3/n goes to zero, and we have:
½ >= C
This is also true for any positive constant C, which means ½n^2 + 3n is Ω(n^2).
Since ½n^2 + 3n is both O(n^2) and Ω(n^2), it is indeed Θ(n^2).

g) n3 + 4n2 =Ω(n2)

To prove that n^3 + 4n^2 is Ω(n^2), we need to show that it is a lower bound for n^2.

In Ω-notation, we need to find constants C and k such that for all values of n greater than or equal to k, the following inequality holds:

n^3 + 4n^2 >= C * n^2

Now, let's simplify the inequality:

n^3 + 4n^2 >= C * n^2

Divide both sides by n^2:

n + 4 >= C

As n approaches infinity, the terms n and 4 both increase. For this inequality to be true, we can choose C to be greater than or equal to 5 (e.g., C = 5).

So, for all values of n greater than or equal to some constant k (which might be 1), n^3 + 4n^2 is indeed a lower bound for n^2 with C = 5.

Therefore, n^3 + 4n^2 is Ω(n^2).

24.Algorithm Sum(n)

S 0

For i 1 to n do

S S + i

Return S

a) What does this algorithm compute?

This algorithm computes the sum of integers from 1 to n. In other words, it calculates the sum of all numbers from 1 to n, inclusive.

b) What it its basic operation?

The basic operation in this algorithm is the addition operation, specifically the addition of 'i' to the variable 'S' on each iteration of the loop.

c) How many times is the basic operation executed?

The basic operation is executed n times because the loop runs from 1 to n, and on each iteration, the basic operation (addition) is performed.

d) What is the efficiency class of this algorithm?

The efficiency class of this algorithm can be analyzed using Big O notation. Since the loop runs from 1 to n and performs a constant-time operation on each iteration, the time complexity of this algorithm is O(n).

Long Answers Questions (THREE, FOUR OR FIVE Marks Questions)

1. *What is an Algorithm? Explain the various criteria for writing an algorithms with example?*

"An algorithm is a sequence of unambiguous instructions for solving the problem and to obtain a required output for any legitimate input in finite amount of time".

• A step-by-step procedure to solve a problem.
• Any special method of solving a certain kind of problem.
• Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
• A sequence of computational steps that transform the input into the output.
• Algorithms are the ideas behind computer programs

Criteria or the properties of algorithm

• An algorithm can take zero or more inputs.
• It should give one or more outputs.
• Definiteness: Each instruction should be clear and unambiguous.
• Finiteness: All cases the algorithm must terminate in finite number of steps.
• Effectiveness: It must be simple and feasible.

2. *Explain Euclid Algorithm with example to find the GCD of two numbers.*

//Algorithm: Euclid(m,n) an algorithm to find gcd of 2 numbers
// Two positive numbers m and n are the input.
// Output: greatest common divisor of m and n.
Step1. If n=0 return the value of m as the answer and stop else proceed to step 2.
Step2. Divide m by n and assign the value of the remainder to r.
Step3. Assign the value of n to m and the value of r to n. Go to step 1

ALGORITHM Euclid(m,n)
// Computes gcd (m,n) by Euclid's Algorithm.
// Input : Two non negative, not both zero integers m & n.
// Output : gcd of m and n.
while n≠0
do r←m mod n
m←n
n←r
return m

3. *Explain Consecutive integer checking methods of find the GCD of two numbers.*

Step1. Assign the value of min{m,n} to t.

Step2. Divide m by t. If the remainder is zero go to step 3, otherwise go to step 4.

Step3. Divide n by t. If the remainder is zero then gcd=t and Stop, otherwise go to step 4.

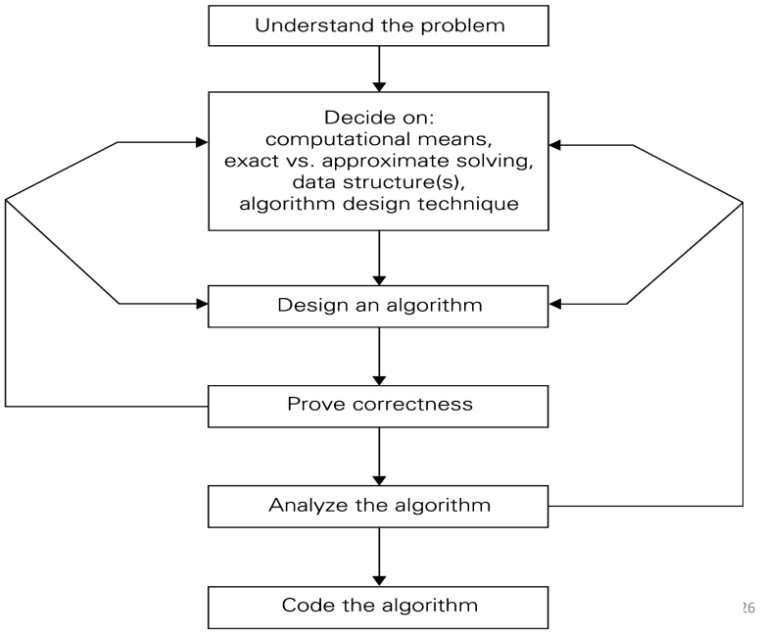
Step4. Decrement the value of t by 1 and go to Step 2.

Disadvantages of Recursive Integer Checking Algorithm

• Integer checking algorithm will not work if one of the inputs is zero.

• So it is required to specify the range of algorithm's input.

4. *Explain Algorithm design and analysis process with flow diagram.*



There are sequence of step in designing and analyzing an algorithm as shown below.

– Understand the problem

• Before designing the algorithm it is important to understand the problem completely. • Read the problem description carefully and ask questions if you have any doubt about the problem. • Better to do small examples by hand, think about special cases and ask questions again if needed. • A correct algorithm is not one, that works most of the time, but correct algorithm is one that works correctly for all legitimate inputs. • It is very important to specify exactly the range of inputs of an algorithm needs to handle.

– Ascertaining the capabilities of a Computational device

Most of the algorithms today are destined to be programmed for a computer closely resembling the von Neumann Machine architecture where instructions are executed sequentially. Such algorithms are called Sequential algorithms. • Some newer computer can execute operations parallel. The algorithms that take advantage of this capability are called Parallel algorithms.

– Choosing between the Exact and Approximate Problem solving

Problem can be solved exactly or approximately. • The algorithms which solve the problem exactly are called exact algorithms. • The algorithms which solve the problem approximately are called approximation algorithms.

– Deciding appropriate data structure

Some algorithms do not demand any particular representation for their input data. – Dynamic programming , Sorting • But some of the algorithm designing techniques depends on structuring data specifying a problem instance. • In object oriented programming data structures are very important for both design and analysis of algorithms.

– Algorithm design techniques

It is a general approach to solve a problem algorithmically that is applicable to a variety of problems from different areas of computing. • They provide guidance for designing algorithms for new problems. • Algorithm design technique makes it possible to classify algorithms according to an underlying idea.

– Methods of Specifying an Algorithm

⎪ Using Natural Language:

♣ This method may leads to ambiguity. ♣ It is very difficult to describe the algorithm clearly. ⎪ Using Pseudo codes:

♣ It is a mixture of natural language and programming language like constructs. ♣ It is more precise than a natural language.

⎪ Using Flow Charts:

♣ It is a method of expressing an algorithm in pictorial manner ♣ It is a method of expressing an algorithm by a collection of connected geometric shapes containing the description of algorithm's step. ♣ This technique is useful for simple algorithms.

– Proving algorithms correctness

Once an algorithm has been specified we need to prove its correctness. • We have to prove that the algorithm yields required result for every legitimate input

in a finite amount of time. • For some algorithms, a proof of correctness is very easy; for others, it can be quite complex. This can be done by doing some validation process. • An algorithm is incorrect if it fails for one instance of input, then you needs to redesign the whole problem.

– Analyzing an Algorithm

⎪ Time efficiency: – It indicates How fast the algorithm runs.

⎪ Space efficiency: – It indicates How much extra space the algorithm needs.

⎪ Simplicity: – Simpler algorithms are easier to understand.

⎪ Generality: – It is easier to design an algorithm for a problem posed in more general terms.

– Coding an algorithm

• Good Algorithm is a result of repeated effort and rework. • After designing a good algorithm it is implemented as computer programs. • After implementing an algorithm the program needs to be optimized to increase the speed of operation. • A working program provides an opportunity to perform the analysis of algorithm. • The analysis is based on the several inputs and then analyzing the results obtained from the program.

5. *Explain any FIVE Problem types.*

Numerical Problems:

• These are the problems that involve mathematical objects of continuous nature.

• For example: Solving equations, computing definite integrals, evaluating functions and so on.

• The majority of such problems can be solved only approximately. These problems require manipulating real numbers, which can be represented approximately in computer.

• Large number of arithmetic operation leads to round off error which can drastically distort the output.

String Processing

• A string is a sequence of characters. Strings can be text strings which consist of letters, numbers and special characters. The bit string contains only zeros and ones.

• A string processing algorithm is used to search a given word in a text.

• String processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

• String matching is one kind of such problem which has many algorithms to solve it.

6. Explain following

a. Graph problem

• One of the oldest area's in algorithms is graph algorithm.

• A graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.

• They can be use for modeling wide variety of real life applications

. • Basic graph algorithm include graph traversal algorithms, shortest path algorithms and topological sorting for graphs with directed edges.

b. Combinatorial problems

• The travelling sales men problem and the graph controlling problem are examples.

• These problems ask to find a combinatorial object such as a permutation, a combination or a subset – that satisfies certain constraints and has some desired property.

• These are the most difficult problems. Because

– The number of combinatorial objects grows extremely fast with a problem's size racing unimaginable magnitude even for moderate sized instances.

– There are no algorithms for solving such problems exactly in an acceptable amount of time.

c. Geometrical problems.

• They deal with geometric objects such as points, lines, and polygons.

• These algorithms are used in developing applications for Computer graphics, robotics and tomography.

• The examples for these problems are

• Closest pair problem

• Convex hull problem

6. *Explain the fundamentals of data structure*

• A data structure can be defined as a particular scheme of organizing related data items.

• Linear Data Structures: – Arrays – Linked Lists

• Graphs

• Trees

• Sets and dictionaries

. 8. *Write a note on Graph data structure*.

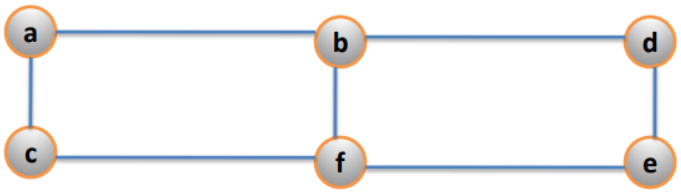
• A graph G=(V,E) is defined by a pair of two sets: a finite set V of items called vertices and asset E of pairs of these items called edges.

• If the pair of vertices is unordered then the graph is undirected.

• Example: vertices (u,v) is as same as (v,u)).

• E={(a,b),(a,c),(b,f),(b,d),(d,e),(f,e),(f,c)}

• V={a,b,c,d,e,f}

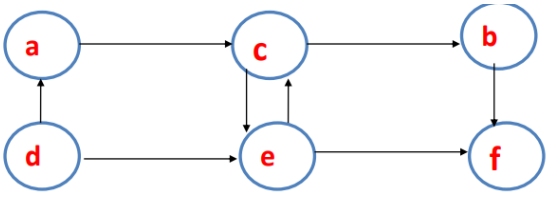


• If the pair of vertices are ordered then the graph is called directed.

• Example: Vertices (u,v) is directed from u to v

• Directed graphs also called as Digraphs.

• Loops: Edges connecting vertices to themselves.



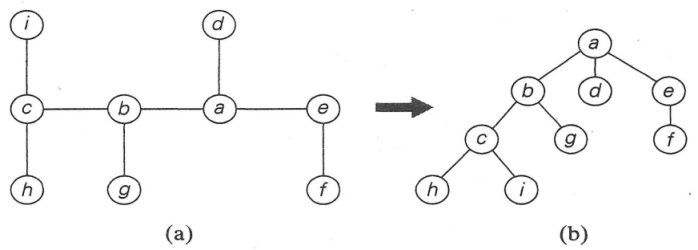
9. Write a note on following data structures.

a. Tree

• It's a connected acyclic graph.

Forest: • It's a graph that has no cycles but not necessarily connected.

Rooted Tree: • In a tree it is possible to select an arbitrary vertex and consider is as the root. Such tree is called as rooted tree.



(a) Free tree. b) Its transformation into a rooted tree.

- It is depicted by placing root on the top.

• Depth of a vertex v is the length of the simple path from the root to v. Height of the tree is the length of the longest simple path.

• For any vertex v in a tree T, all the vertices on the simple path from the root to that vertex are called ancestors of v.

• If (u, v) is the last edge of the simple path from the root to vertex v (and u /= i>), u is said to be the parent of v and v is called a child of u.
• Vertices that have the same parent are said to be siblings.

• A vertex with no children is called a leaf.

• Vertex with at least one child is called parental.

• All the vertices for which a vertex v is an ancestor are said to be descendants of v.

• A vertex v with all its descendants is called the subtree of T rooted at that vertex.

b. Sets

• A set can be described as an unordered collection (possibly empty) of distinct items called elements of the set.

• Ex: S = {2,3,5,7})

• Computer Implementation:

• 2 methods:

• The first considers only sets that are subsets of some large set U called the universal set. If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a bit vector. – U = {1,2,3,4,5,6,7,8,9}, then

• S = {2, 3, 5, 7}

• Bit String 011010100.

• Using linked list structure to indicate the sets elements. His can be only used for finite sets.

c. Dictionary

• A Data structure that implements – searching of a given item, adding a new item, and deleting an item, is called as Dictionary.

Abstract Data Types: • A set of abstract objects representing data items with a collection of operations that can be performed on them.

10.Explain Space complexity and Time complexity with example.

• Space efficiency:

• The extra space the algorithm requires.

• Depends on – data space, program space, stack space.

• Program space:

– The space required for storing the machine program generated by the compiler or assembler.

• Data space:

– The space required to store the constants or variables.

• Stack space:

– The space required to store the return address along with parameter passed to the function

Time efficiency:

• How fast an algorithm runs.

• Depends on – speed of the computer, choice of the programming language, compiler used, choice of the algorithm, size of input and outputs etc.

• No control over the speed of the computer and programming language

1) Basic operation Count

2) Step Count

3) Asymptotic Notation

Units of measuring Running Time

• Use physical unit of time i.e. second, Milliseconds.

• The running time depends on the – Compiler in developing the machine instruction. – The clock speed of the system.

• Since we are dealing with efficiency of the algorithm, it should be independent of all these above factors.

11.Write an algorithm find sum of two matrixes

also calculate its time complexity.

ALGORITHM MatrixMultiplication (A[0..n-1,0..n-1], B[[0...n1,0..n-1]

//Multiplies two square matrices of order n by the definitionbased algorithm

//Input: Two n-by-n matrices A and B

//Output: Matrix C=AB

for i ← 0 to n-1

do

for j ← 0 to n-1

do C[i,j+]← 0

for k ← 0 to n-1

do C[i,j+] ← C[i,j] + A[i,k] * B*[k,j]

Return C

Calculate the mathematical analysis of Matrix Multiplication

• T (n) ≈ cop C (n) for multiplication T (n) ≈ cop n 3 similarly for Addition

• T (n) ≈ cop C (n) for Addition T (n) ≈ cop n 3 So T (n) ≈ n3 (Cm + Ca)

12.Write an algorithm find the sum of n numbers also calculate its space and time complexity.

#include Void main()

{

int x, y, z, sum;

Printf("Enter the three numbers");

Scanf("%d, %d, %d", &x, &y, &z);

Sum= x + y + z;

Printf("the sum = %d", sum);

- • In the above program the space needed by x, y, z and sum is independent of instance characteristics. The space for each integer variable is 2.

• We have 4 integer variables and space needed by x, y, z and sum are 4 * 2 = 8 bytes.

– S(p) = Cp + Sp – S(p) = 8 + 0 = 8

13. Explain the following w.r.t algorithm efficiency.

a. Measuring input size

• Almost all algorithms run longer on larger inputs

. • For sorting larger arrays, multiply larger element matrix it takes longer time.

• The same is in the case of spell checking algorithm, where we have to check each character which depends on the number of character present.

• For such algorithms, we need to prefer measuring the input size by the number b of bits in the n's binary representation: b = [log 2 n] + 1

b. Unit for measuring the run time

• Use physical unit of time i.e. second, Milliseconds.

• The running time depends on the

– Compiler in developing the machine instruction.

– The clock speed of the system.

• Since we are dealing with efficiency of the algorithm, it should be independent of all these above factors.

c. Order growth

• Consider n is input size of the algorithm.

• Some algorithms work faster for all values of n.

• But some algorithms execute faster for smaller values of n. As the values of n increases, they tend to be slow.

• The behavior of some algorithm changes with increase in value of n.

• This change in behavior as the value of n increases is called order of growth.

• The table helps to compare among all the functions.

• The function growing the slowest among these is the logarithmic function. (log2n)

• It grows so slowly, in fact, that we should expect a program implementing an algorithm with a logarithmic basic-operation count to run instantaneously on inputs of all realistic sizes

. • On the other end of the exponential function 2 n and the factorial function n!, both these functions grow so fast that their values become astronomically large even for rather small values of n.

• So the function 2 n and n! are called as exponential growth functions.

14.Explain Worst case, Best case and average case with example

– The best-case efficiency of an algorithm for the input size n for which the algorithm takes least time or the fastest among all possible inputs of that size.

– Cbest (n) = 1

– Searching key may be present in the very first location itself.

– The worst-case efficiency of an algorithm is its efficiency for input of size n, for which the algorithm runs the longest among all possible inputs of that size.

– Cworst (n) = n

– When there are no matching elements.

– Largest number of key comparison among all possible inputs of size n.

– The average case efficiency of an algorithm is its efficiency for the random input of size n.

– Cavg (n) = p(n+1)/2 +n(1-p)

– Searching element may be present at some where in the middle.[example next]

15.Write an algorithm to perform sequential search and also calculate its Worst case, Best case and average case complexity.

Straight forward algorithm that searches for a given item in a list of n elements, by checking successive elements of the list until either a match with the search key is found or the list is exhausted

ALGORITHM Sequential Search (A [0..n - 1], K)

// Searches for a given value in a given array by sequential search

// Input: An array A[0..n — 1] and a search key K

// Output: Returns the index of the first element of A that matches K or returns -1 if there are no matching elements.

i ←0

while i < n and A[i] ≠ K

do i ← i + 1

if i<n return i

else return -1

Worst case:n

Best case:1

Average case: (1+2+3+…..+n)/n=n(n+1)/2n=(n+1)/2

16.Explain Big O notation with example.

• It is a formal method of expressing the upper bound of an algorithm's running time.

• Measures the longest amount of time it could possibly take for the algorithm to complete.

• There is no lower bound for t(n) for large values of n. The lower bound can be obtained using Big Omega.

Definition: A function t(n) is said to be in O(g(n)), denoted t(n) Є O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e.,

if there exist some positive constant c and some nonnegative integer n0 such that

t(n) <= c g(n) for all n >= n0[example down solved]

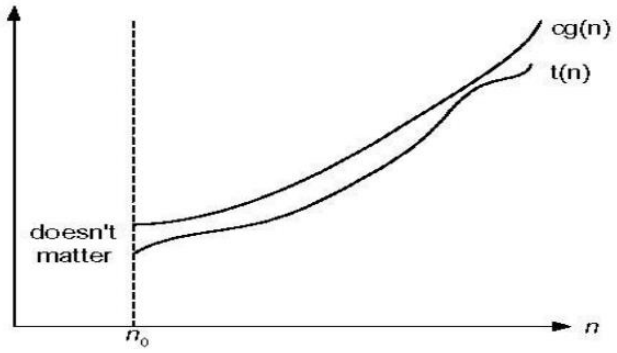


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

17.Explain Big Omega notation with example.

A function t(n) is said to be in Ω(g(n)), denoted t(n) Є Ω(g(n)), if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that:

t(n) ≥ cg(n) for all n >= n0 .[example solved down]

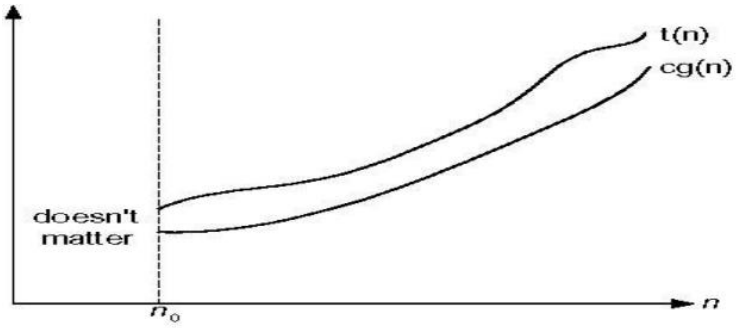


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

18. Explain Big Theta notation with example.

A function t(n) is said to be in Θ (g(n)), denoted t(n) Є Θ(g(n), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

c2 g(n) ≤ t(n) ≤ c1 g(n) for all n ≥ n0

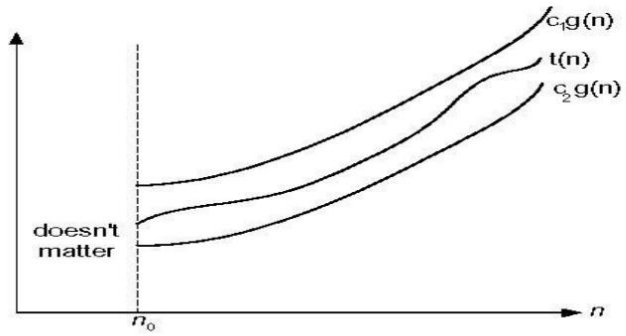


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

19.Explain asymptotic notations Big O, Big Ω and Big θ that are used to compare the order of growth of an algorithm with example.

[Explained]

20.Define Big O notation and prove

e a) 100n + 5=O(n2)

– 100n + 5 ≤ 100n + n

= 101n ≤ 101n2

for all n ≥ 5 n0 = 5 and c = 101 is a solution

– 100n + 5 ≤ 100n + 5n

= 105n ≤ 105n2

for all n ≥ 1

n0 = 1 and c = 105 is also a solution

Must find SOME constants c and n0 that satisfy the asymptotic notation relation

b) 5n2 + 3n + 20 = O(n2)

The equation 5n^2 + 3n + 20 is indeed of the order O(n^2) in terms of its growth rate as n approaches infinity. In big O notation, we focus on the dominant term with the highest power of n, and the coefficient is not significant.
In this case, as n approaches infinity, the n^2 term dominates the growth rate, and the other terms (3n and 20) become less significant. So, we can say that 5n^2 + 3n + 20 is in O(n^2).
This is a simplified way to prove that it's O(n^2) based on the dominant term, which represents the growth rate of the function as n gets larger.

c) n 2+n = O(n3)

To prove that the function f(n) = n^2 + n is in O(n^3), we need to show that there exist constants C and n_0 such that for all n greater than or equal to n_0, f(n) is bounded above by C * n^3.
Let's analyze the function f(n) = n^2 + n:
f(n) = n^2 + n
We need to find C and n_0 such that:
n^2 + n <= C * n^3 for all n >= n_0
Now, let's try to find suitable C and n_0:
n^2 + n <= C * n^3
1 + 1/n <= C * n
Now, if we choose C = 2 and n_0 = 1, then for all n >= 1:
1 + 1/n <= 2n
Since 1 + 1/n is always less than or equal to 2n for n >= 1, we have shown that f(n) = n^2 + n is bounded above by C * n^3 for n greater than or equal to n_0.
Therefore, f(n) = n^2 + n is indeed in O(n^3).

d) 3n+2=O(n)

The statement that 3n + 2 is in O(n) is not correct. In big O notation, we are looking for an upper bound on the growth rate of a function, and we want to find a constant C and a value n_0 such that for all n greater than or equal to n_0, the function is bounded above by C * n.
In the case of 3n + 2, no such constant C and value n_0 can be found. This function is actually in O(n), not O(1). It grows linearly with n, and you can choose C = 4 and n_0 = 1 as an example where:
3n + 2 <= 4n for all n >= 1.
So, 3n + 2 is in O(n), not O(1).

e) 1000n2+100n-6=O(n2)

To prove that the function f(n) = 1000n^2 + 100n - 6 is in O(n^2), you need to show that there exist constants C and n_0 such that for all n greater than or equal to n_0, f(n) is bounded above by C * n^2.
Let's analyze the function f(n) = 1000n^2 + 100n - 6:
f(n) = 1000n^2 + 100n - 6
We need to find suitable C and n_0 such that:
1000n^2 + 100n - 6 <= C * n^2 for all n >= n_0
Now, let's try to find such constants:
1000n^2 + 100n - 6 <= 1000n^2 + 100n (since -6 is a constant)
If we choose C = 1001 and n_0 = 1, then for all n >= 1:
1000n^2 + 100n - 6 <= 1000n^2 + 100n (which is less than or equal to 1001n^2 for n >= 1)
Therefore, f(n) = 1000n^2 + 100n - 6 is indeed in O(n^2) with C = 1001 and n_0 = 1.

16. Define Big Omega notation and prove

a) n 3∈Ω (n2)

The statement that n^3 ∈ Ω(n^2) is true. In big Omega notation (Ω), we are looking for a lower bound on the growth rate of a function, and we want to find constants C and n_0 such that for all n greater than or equal to n_0, the function is bounded below by C * n^2.
In this case, n^3 is indeed a higher-order polynomial than n^2. To prove this, you can set C = 1 and n_0 = 1. For all n greater than or equal to 1, you have:

n^3 >= n^2

Therefore, n^3 is bounded below by C * n^2 (where C = 1 and n_0 = 1), which means n^3 is in Ω(n^2).

b) Prove that 2n + 3 = Ω (n)

To prove that 2n + 3 = Ω(n), we need to show that there exist constants C and n_0 such that for all n greater than or equal to n_0, the function is bounded below by C * n.
In this case, it's relatively straightforward. We can choose C = 1 and n_0 = 1. For all n greater than or equal to 1:
2n + 3 >= n
Therefore, 2n + 3 is bounded below by C * n (where C = 1 and n_0 = 1), which means 2n + 3 is in Ω(n).

c) $\frac{1}{2}n(n-1) \in \Omega(n^2)$.

To prove that 1/2n(n-1) ∈ Ω(n^2), we need to show that there exist constants C and n_0 such that for all n greater than or equal to n_0, the function is bounded below by C * n^2.
Let's analyze the function 1/2n(n-1):
1/2n(n-1) = 1/2 * (n^2 - n)
Now, we want to find C and n_0 such that:
1/2 * (n^2 - n) >= C * n^2 for all n >= n_0
If we choose C = 1/4 and n_0 = 1, then for all n greater than or equal to 1:
1/2 * (n^2 - n) >= 1/4 * n^2
This is true because n^2 - n is always greater than or equal to 1 for n greater than or equal to 1.
So, we've found the constants C = 1/4 and n_0 = 1, and the function 1/2n(n-1) is indeed in Ω(n^2).

d) Prove that n3 + 4 n2 = Ω (n2)

To prove that n^3 + 4n^2 = Ω(n^2), we need to show that there exist constants C and n_0 such that for all n greater than or equal to n_0, the function is bounded below by C * n^2.

In this case, it's quite straightforward. You can choose C = 1 and n_0 = 1. For all n greater than or equal to 1:
n^3 + 4n^2 >= n^2

Therefore, n^3 + 4n^2 is bounded below by C * n^2 (where C = 1 and n_0 = 1), which means n^3 + 4n^2 is in Ω(n^2).

17. Define Big Theta notation and prove

a) n2+5n+7 = Θ (n2)

To prove that n^2 + 5n + 7 = Θ(n^2), we need to show two things:
1. n^2 + 5n + 7 = O(n^2): This means the function is bounded above by a constant times n^2.
2. n^2 + 5n + 7 = Ω(n^2): This means the function is bounded below by a constant times n^2.
Let's start with the first part:
1. n^2 + 5n + 7 = O(n^2):
To prove this, we need to find constants C and n_0 such that for all n greater than or equal to n_0:
n^2 + 5n + 7 <= C * n^2
If we choose C = 13 (which is greater than the coefficients of the terms in the equation) and n_0 = 1, then for all n greater than or equal to 1:
n^2 + 5n + 7 <= 13n^2
So, the first part is proven.

b) 1/2n2 +3n =Θ (n2)

To prove that 1/2n^2 + 3n = Θ(n^2), we need to show two things:
1. 1/2n^2 + 3n = O(n^2): This means the function is bounded above by a constant times n^2.
2. 1/2n^2 + 3n = Ω(n^2): This means the function is bounded below by a constant times n^2.
Let's start with the first part:
1. 1/2n^2 + 3n = O(n^2):
To prove this, we need to find constants C and n_0 such that for all n greater than or equal to n_0:
1/2n^2 + 3n <= C * n^2
If we choose C = 5 and n_0 = 1, then for all n greater than or equal to 1:
1/2n^2 + 3n <= 5n^2
So, the first part is proven.
2. 1/2n^2 + 3n = Ω(n^2):
To prove this, we need to find constants C and n_0 such that for all n greater than or equal to n_0:

1/2n^2 + 3n >= C * n^2
If we choose C = 1/4 (which is less than the coefficient of the n^2 term in the equation) and n_0 = 1, then for all n greater than or equal to 1:
1/2n^2 + 3n >= 1/4n^2
So, the second part is proven.
Since we've proven both parts, we can conclude that 1/2n^2 + 3n = Θ(n^2).

18. Explain with example mathematical analysis of non-recursive algorithm.

1. Decide on a parameter(s) indicating an input's size.

2. Identify the algorithm's basic operation.

3. Check whether the number of times the basic operation is executed depends only on the input size. If it also depends on some additional property, determine the worst-case, average-case, and best-case complexities separately.

4. Find out C(n) [the number of times the algorithm's basic operation is executed.]

5. Using standard formulas establish the order of growth.

Example: Linear Search Algorithm. (already discussed)

19. Write an algorithm to Find the largest element in an array and also perform mathematical analysis.

ALGORITHM Maxelement(A*0….n-1])

//Determines the value of the largest element in a given array.

//Input: An array A*0…..n-1]

//Output: The value of the largest element in A.

maxval ←A[0]

for i ← 1 to n-1

do

if A[i]>maxval

maxval ← A[i]

return maxval

• n is the input number.

• Basic operation - if A[i]>maxval

• Each time control enters into the loop, basic operation is executed once. So C(n)=n-1.

• For any input size of n, the algorithm executes in n-1 comparisons to determine the largest value. Hence there is no need to determine best, average and worst case separately. So expressed using Θ notation.

• The comparison statements are executed exactly once for each iteration of i ranging from 1 to n-1. Therefore C(n) = $\sum_{i=1}^{n-1} 1 = n - 1 \in \theta(n)$

• Result = Upper bound – Lower bound + 1

20. Write an algorithm to Checking for Unique elements in an array and also perform mathematical analysis.

ALGORITHM UniqueElements (A[0..n - 1])

//Checks whether all the elements in a given array are distinct

//Input: An array A [0..n - 1]

//Output: Returns "true“ if A contains distinct elements, otherwise

//Returns "false“.

for i ← 0 to n — 2

do

for j ← i + 1 to n – 1

do

if A[i] = A [j]

return false

return true

• Calculate the mathematical analysis.

• We get C(n) = n2 /2 - n/2 ≤ n2

for n≥ 0 Є Θ (n 2)

• The natural measure of the input's size here is the number of elements in the array. i.e, n

• Since inner most loop contains a single operation, it is considered as the algorithm's basic operation.

• The basic operation depends on the size as well as positions where the same or the equal elements are placed.

• So worst case can be consider in 2 cases. ϖWhen all input elements are different ϖWhen equal elements are occupied at the last 2 positions.

21. Write an algorithm to perform matrix multiplication and also perform mathematical analysis.

ALGORITHM MatrixMultiplication (A[0..n-1,0..n-1], B[[0...n1,0..n-1]

//Multiplies two square matrices of order n by the definitionbased algorithm

//Input: Two n-by-n matrices A and B

//Output: Matrix C=AB

for i ← 0 to n-1

do for j ← 0 to n-1

do C[i,j] ← 0

for k ← 0 to n-1

do C[i,j] ← C[i,j] + A[i,k] * B[k,j]

Return C

• Calculate the mathematical analysis of Matrix Multiplication

• T (n) ≈ cop C (n) for multiplication T (n) ≈ cop n 3 similarly for Addition

• T (n) ≈ cop C (n) for Addition T (n) ≈ cop n 3 So T (n) ≈ n3 (Cm + Ca)

Analysis of Matrix Multiplication

• size –n- order of matrix.

• Multiplication and addition are considered as basic operation.

• Execution of basic operation depends on the input size.

• Basic operation is executed exactly once for entire range of i, j, k

• Total number of times the multiplication statement is executed can be calculated. ie n 3

• Total number of times the addition statement is executed can be calculated. ie n 3

• So total time efficiency T(n) = mop M(n)+aop A(n) = n 3 (mop+aop)

22. Write anon-recursive algorithm to Count the number of bits in a number. And also perform mathematical analysis.

• Decide on a parameter/s indicating an input's size.

• Identify the algorithm's basic operation.

• Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.

• Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

• Solve the recurrence or at least ascertain the order of growth of its solution.

23. List the steps for analyzing the time efficiency of recursive algorithm.

Analyzing the time efficiency of a recursive algorithm involves assessing its time complexity, which describes how the algorithm's runtime grows in relation to the size of the input. To analyze the time efficiency of a recursive algorithm, you can follow these steps:

1. Define the recursive algorithm: Understand the recursive algorithm's structure and how it breaks down the problem into smaller subproblems. Identify the base case(s) and the recursive case(s).

2. Write the recurrence relation: Express the algorithm's time complexity using a recurrence relation. The recurrence relation describes the algorithm's runtime in terms of the size of the input and the time complexity of the recursive calls.

3. Determine the base case: The base case(s) represent the simplest input(s) that can be directly solved without further recursion. Identify the number of operations required to solve the base case(s).

4. Analyze the recursive case: Determine how many recursive calls are made, the size of the subproblems in each call, and the additional work done outside of the recursive calls. This step will involve breaking down the problem's size and analyzing the work required for each subproblem and the combined work.

5. Express the recurrence relation: Write the recurrence relation that relates the time complexity of the algorithm to the time complexity of its recursive calls. It's usually in the form $T(n) = f(n) + T(g(n))$, where $T(n)$ is the time complexity of the algorithm for an input of size n, $f(n)$ represents the work done outside of the recursive calls, and $T(g(n))$ represents the sum of the time complexities of the recursive calls.

6. Solve the recurrence relation: Depending on the form of the recurrence relation, you may need to use mathematical techniques to solve it. Common methods include substitution, iteration, and the Master Theorem, which helps determine the time complexity in terms of big O notation.

7. Express the time complexity in big O notation: After solving the recurrence relation, express the time complexity of the recursive algorithm in big O notation, which provides an upper bound on the algorithm's growth rate concerning the input size.

8. Perform time complexity analysis: Assess the derived time complexity to understand how the algorithm's runtime scales with the input size. This analysis helps you determine whether the algorithm is efficient for the problem at hand and compare it to other algorithms.

9. Verify with empirical testing: If needed, you can verify the time complexity analysis by conducting empirical testing, such as measuring the algorithm's actual runtime on various inputs and comparing it to the expected time complexity.

By following these steps, you can systematically analyze the time efficiency of a recursive algorithm and gain insights into its performance characteristics.

23. Explain with example mathematical analysis of recursive algorithm.

Let's illustrate the mathematical analysis of a recursive algorithm with a classic example: the computation of the Fibonacci sequence using a recursive algorithm. The Fibonacci sequence is defined as follows: F(0) = 0, F(1) = 1, and F(n) = F(n-1) + F(n-2) for n > 1.

Here's the recursive algorithm to compute the nth Fibonacci number:

```python
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

We'll analyze the time efficiency of this algorithm using a mathematical approach.

1. Define the recursive algorithm:

   - The Fibonacci recursive algorithm computes the nth Fibonacci number by recursively calling itself with smaller values of n until it reaches the base cases (n = 0 or n = 1).

2. Write the recurrence relation:

   - The recurrence relation for the time complexity of this algorithm is T(n) = T(n-1) + T(n-2) + O(1), where T(n) represents the time complexity of computing the nth Fibonacci number, and the O(1) term accounts for the constant time operations within each function call.

3. Determine the base case:

- The base cases for this algorithm are n = 0 and n = 1. In both cases, the algorithm returns a constant value, and thus the work required is O(1).

4. Analyze the recursive case:

- For n > 1, the algorithm makes two recursive calls: one with n-1 and another with n-2. The time complexity of these calls is T(n-1) and T(n-2), respectively. Additionally, the algorithm performs constant-time operations for addition and the conditional statement (O(1)).

5. Express the recurrence relation:

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

6. Solve the recurrence relation:

- Solving this recurrence relation can be done using various techniques, such as substitution, iteration, or the Master Theorem. In the case of the Fibonacci algorithm, it results in an exponential time complexity.

7. Express the time complexity in big O notation:

- The time complexity of the Fibonacci recursive algorithm is O(2^n). This means that the algorithm's runtime grows exponentially with the input size, making it highly inefficient for large values of n.

8. Perform time complexity analysis:

- The analysis indicates that the Fibonacci recursive algorithm is not efficient, especially for large values of n. The exponential time complexity results in a rapidly increasing number of function calls and computations.

To summarize, the mathematical analysis of the recursive Fibonacci algorithm shows that it has an exponential time complexity, which makes it impractical for computing Fibonacci numbers for large values of n. In such cases, a more efficient approach, such as memoization or dynamic programming, is preferred to reduce the number of redundant calculations and improve performance.

24. Write an algorithm to find the factorial of a number using recursion and also perform mathematical analysis.

• The factorial function F(n)=n! for an arbitrary nonnegative integer.

• n!= 1….(n-1).n = (n-1)!.n for n≥1.

ALGORITHM F(n)

//Computes n! recursively.

//Input: A non-negative integer.

//Output: The value of n!

If n=0

return 1

else

return F(n-1) * n

Analysis of factorial function

• ‘n’ is the input size of this algorithm.

• The basic operation of the algorithm is multiplication.

• Number of execution (multiplication) is denoted as M(n).

• The stopping condition - If n=0 return 1 which serves two purposes – No more multiplications. – The function call stops.

• For solving recursive relations, we use the technique called method of backward substitutions.

– M(n) = 0

if n = 0 – M(n)

= 1 + M(n-1) for n > 0

• Here 1 : to multiply f(n-1) and n M(n-1) :

to compute f(n-1)

M(n) = M(n-1) + 1

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

= M(n-2) + 2

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

= M(n-3) + 3 …

= M(n-i) + i

(In general) = M(0) + n (By taking the advantage of the initial condition given) = n (n times multiplication happens)

• The method is called backward substitution

25, Write an algorithm to perform Towers of Hanoi using recursion and also perform mathematical analysis.

• In this puzzle, we have 'n' disks of different sizes and 3 pegs.

• Initially all the disks are on the first peg in order of size, the largest on the bottom and the smallest on the top.

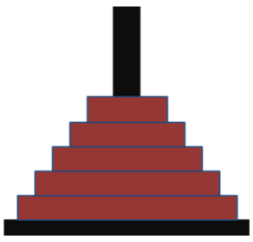
• The goal is to move all the disks to 3 rd peg using second one as auxiliary, if necessary.

• The condition is we can move only one disk at a time.

• The larger disk should be below the smaller disk.

• tower(int n, int source, int temp, int destination)

```
ALGORITHM Tower of Hanoi (n, x, y, z)
//Input: n, number of disks
//Output: Order of movements
If (n>=1) then
Begin
   Tower of Hanoi(n-1,x,y,z);
   Write ("Move disk from", x, "to", y);
   Tower of Hanoi(n-1,z,y,x);
```



Time efficiency

• The number of disks 'n' is the input's size indicator.

• Moving a disk is the algorithm's basic operation.

• The number of moves M(n) depends on the 'n' value.

• M(n-1) – number of disc movements from source to temp.

• 1 - number of disc movements from source to destination.

• M(n-1) – number of disc movements from temp to destination.

• Recurrence equation is M(n) = M(n-1) + 1 + M(n-1) for n>1 M(n) = 2M(n-1) + 1

• If n=1, we can simply move the single disk directly from the source peg to the destination peg.

• So Initial condition is, M(1) = 1

To solve this we use the backward substitution method:

M(n) = 2M(n-1) + 1.

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

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

2 [2M(n-3)+ 1] +2 + 1 =

2 3M(n-3)+ 2 2+ 2 + 1

• Generally after i substitution we get M(n)= 2 iM(n-i) + 2 i-1+2 i-2+……. +2+1 =

2 iM(n-i) + 2 i – 1

• Since the initial condition is specified for n=1, which is achieved for i= n-1,

we get the following formula.

M(n) = 2 n-1M(n-(n- 1)) +2 n-1 – 1

M(n)= 2 n-1M(1) +2 n-1 – 1

M(n)= 2 n-1+2 n-1 – 1

M(n)=2. 2 n-1 – 1

M(n)= 2 n – 1

• Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of n

Analysis:

- Here the size is n.
- **Basic operation: Addition.**
- When n=1 no additions required. **So A(n) = 0**
- So the recurrence relation is

  A(n) = A(n/2) +1 for n > 1

  A(n) = 0 for n = 1
- **Solving for [n/2] using back tracking becomes very difficult on values of n are not power of 2. So we assume that $n = 2^k$**

A(*n*) = A([n/2]) + 1, and the initial condition is $A(2^0)= A(1)=0$

According to our assumptions put $n= 2^k$

$A(2^k) = A(2^{k-1}) + 1$ **(now apply the back tracking method)**

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

$= A(2^{k-2}) + 2$

$= A(2^{k-i}) + i$ (in general)

Now to get the initial condition put **i = k**

$= A(2^{k-k}) + k$

$= 0 + k$

$n= 2^k$ and hence $k= \log_2 n$

So **$A(n) = \log_2 n \in \Theta(\log n)$**

26. State the recursive algorithm to count the bits of a decimal number in its binary representation. Give its mathematical analysis.

27.Consider the following algorithm.

```
Algorithm GUESS (A[ ] [ ])
for i ← 0 to n – 1
    for j ← 0 to i
        A [i] [j] ← 0
```

i) What does the algorithm compute?

ii)What it its basic operation?

iii) What is the efficiency of this algorithm?

27. Solve the following recurrence relation.

a) x(n) = x(n-1) + 5 for n >1 , x(1) =0 b) x(n) = 3x(n-1) for n > 1 x(1)=4 c).x(n)=x(n-1)+n for n>1 x(0)=0 28. Find the time complexity of below algorithm.

```
algorithm Fibonacci(n)
{
  if n <= 1 then
    output 'n'
  else
    f2 = 0;
    f1 = 1;

    for i = 2 to n do
    {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
    }
    output 'f'
}
```

Unit-2

2 marks Questions

1. What is Brute force approach of problem solving? Give a example.

:-It is a straight forward approach to solving a problem, usually directly based on the problems statement and definitions of the concepts involved.

• Example:

1. Computing a n
2. Computing n!
3. Sequential search

2. List any four importance of Brute force.

:- The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.

It is a generic method and not limited to any specific domain of problems.

The brute force method is ideal for solving small and simpler problems.

It is known for its simplicity and can serve as a comparison benchmark.

3. Write the number of operations and time complexity of selection sort.

:- Best-case: O(n2), best case occurs when the array is already sorted. (where n is the number of integers in an array)

Average-case: O(n2), the average case arises when the elements of the array are in a disordered or random order, without a clear ascending or descending pattern.

Worst-case: O(n2), The worst-case scenario arises when we need to sort an array in ascending order, but the array is initially in descending order.

4. Write the number of operations and time complexity of bubble sort.

:- Bubble Sort compares the adjacent elements. Hence, the number of comparisons is (n-1) + (n-2) + (n-3) +.....+ 1 = n (n-1)/2 nearly equals to n 2 Hence, Complexity: **O (n 2**) Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is n*n = n 2

5. What do you mean by Sequential Search? Write its time complexity

:-Straight forward algorithm that searches for a given item in a list of n elements, by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

6. Write the worst case and average case complexity of Brute force String Matching.

:-In the worst case after doing 'm' comparisons we willcome to know there is a mismatch. So shift one position in the text. This will happen for all the n-m+1 trails.

• C Worst(m,n)= (n-m+1) m

=mn-m2+m

(mn is higher when compared to m2 because normally n is greater than m)

= mn Є (mn)

• The average case efficiency is = Θ(n)

7. Write the number of operations and time complexity of Closest-Pair Problem.

:- Time Complexity Let Time complexity of above algorithm be T(n). Let us assume that we use a O(nLogn) sorting algorithm. The above algorithm divides all

points in two sets and recursively calls for two sets. After dividing, it finds the strip in O(n) time, sorts the strip in O(nLogn) time and finally finds the closest points in strip in O(n) time. So T(n) can expressed as follows

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

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

T(n) = T(n x Logn x Logn)

8. Define Convex and Convex hull.

:-

1. **Convex:**
   - A set of points in a Euclidean space is considered convex if, for any pair of points within the set, the line segment connecting those two points lies entirely within the set. In other words, a set is convex if, for any two points A and B in the set, all the points on the straight line segment between A and B are also in the set.
2. **Convex Hull:**
   - The convex hull of a set of points is the smallest convex polygon that encloses all of those points. It is like finding the outer boundary of the set. The convex hull is a fundamental concept in computational geometry and is used in various applications, including computer graphics, geographic information systems, and robotics.

9. List any two example algorithms of the brute force approach.

:-1. Computing a n
2. Computing n!
3. Sequential search

10.What do you mean by convex hull problem? Write its time complexity

:-**Convex Hull:**
   - The convex hull of a set of points is the smallest convex polygon that encloses all of those points. It is like finding the outer boundary of the set. The convex hull is a fundamental concept in computational geometry and is used in various applications, including computer graphics, geographic information systems, and robotics.

11.Define Exhaustive search.mechansim.Give example.

:-Its is simply a brute force approach for combinatorial problems.

• Method:

– List all the potential solutions to the problem.

• No repetition of solution.

– Evaluate solutions one by one.

• Discard infeasible solutions and keep track of the best solution found so far.

– When search ends you will get optimal solution.

Example: Traveling Salesman Problem [TSP]

Knapsack Problem

Assignment Problem

12.What do you mean by Knapsack Problem? Write its time complexity

:-Given n items with the corresponding weights w1,w2….wn and values v1,v2,…vn and a knapsack of capacity W.

• Find the most valuable subset of the items that fit into the knapsack.

• Exhaustive search is used to solve the problem.

13.What do you mean by Travelling Salesman Problem? Write its time complexity.

:-The problem is to find the shortest route through a given set.

• Given n cities with known distances between each pair.

• Find the shortest tour or path that passes through all the cities exactly once before returning to the starting city.

• Alternatively: find the shortest Hamiltonian circuit in a weighted connected graph.

14.What do you mean by Job assignment problem? Write its time complexity.

:-There are ‘m’ people who need to assign ‘n’ jobs.

• Each person is assigned to a single job and vice versa

• A cost matrix indicates the cost that is incurred for performing the job j i by the person pi which is denoted as cij

• This problem is to find the minimum total cost.

• Number of combinations for n jobs are n!

Long Answer Questions (THREE, FOUR OR FIVE Marks Questions)

1. *Write an algorithm to sort N numbers using Selection sort. Derive the number of operations and time complexity.*

Selection Sort • We start selection sort by scanning the entire list to find its smallest element and exchange it with the first element, putting the smallest element in its first position in the sorted list. • Then we scan the list, starting with the second element, to find the smallest among the n-1 elements and exchange it with the second element. And so on….

Algorithm: SelectionSort(A[0….n-1])

// Sorts given array using selection sort.

// Input: An array A[0…..n-1] of orderable elements.

// Output: An array A[0…..n-1] sorted in ascending order.

for i←0 to n-2 do

min ← i

for j←i+1 to n-1 do

if A[j]<A[min]

min<-j

swap A[i] and A[min]

Analysis

Input size: number of elements n.

Basic operation: Comparison A[j] < A[min]

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

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

$\in \Theta(n^2)$

2. *Write an algorithm to sort N numbers by applying Bubble sort. Derive the number of operations and time complexity.*

• Bubble sort is a brute-force application to the sorting problem

. • Adjacent elements of the list are compared and exchanged if they are not in order.

• By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. • In the next pass bubbles up the second largest element, and so on… until, after n-1 passes, the list is sorted.

• In bubble sort Largest number is bubbling to the last position. • In selection sort after every iteration small number comes in the first position

Analysis:

Its input size = n

Basic operation: Comparison A[j+1] < A[j]

The number of times it is executed depends only on the arrays size.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$
$$= \sum_{i=0}^{n-2} (n-1-i)$$
$$= \frac{n(n-1)}{2}$$
$$\in \Theta(n^2).$$

```
ALGORITHM BubbleSort(A[0...n-1])
//Sorts the array using bubble sort.
//Input: An array A[0....n-1] of orderable elements.
//Output: An Array A[0....n-1] in ascending order.
for i←0 to n-2 do
    for j←0 to n-2-i do
        if A[j+1] < A[j]
            swap A[j+1] and A[j]
```

3. *Write and describe the Sequential search algorithm.*

• Straight forward algorithm that searches for a given item in a list of n elements, by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

```
ALGORITHM Sequential Search (A [0..n - 1], K)
// Searches for a given value in a given array by sequential
search
// Input: An array A[0..n — 1] and a search key K
// Output: Returns the index of the first element of A that
matches K or returns -1 if there are no matching elements.
i ←0
while i < n and A[i] ≠ K do
        i ← i + 1
    if i<n              return i
        else            return -1
```

4. *Write and describe Brute force String Matching Algorithm*

• Searching for a given word in a text is called as string matching.

• Given a string of 'n' characters called the text

• A string of 'm' characters (m≤n) called pattern

• Find a substring of the text that matches the pattern.

• Align the pattern against the first 'n' characters of the text

• Start matching the corresponding characters from left to right until either all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

• If the character is not matched when compared to the pattern, then the pattern is shifted one position towards the right and comparisons are resumed.

• The process is continued until the end of the text has been reached.

• If the pattern is matched with the substring of the text, then the algorithm returns starting position of the substring in the given text otherwise, the algorithm returns value -1.

• We do the comparison up to n-m position beyond that position there are not enough characters that match the entire pattern.

```
ALGORITHM BruteForceStringMatch(T[0...n-1],P[0...n-1])
//Implements String matching
//Input: Text array T of n characters, and pattern array P of r
   characters.
//Output: Position of first character of pattern if successful otherwise -1
for i←0 to n - m do
   j←0
   while j<m and P[ j ] = T [ i + j ] do
        j←j+1
   if j=m
        return i
return -1
```

Tracing of String Matching

Text : "WAIT AND WATCH"

Pattern : "WAT"

n = 14

j = m = 3

✓ Pattern P is present in the text T starting at position 9.

✓ Total number of times the basic operation executed = 14

i	j	P[j]	T[i + j]
0	0	W	W
	1	A	A
	2	T	I
1	0	W	A
2	0	W	I
3	0	W	T
4	0	W	
5	0	W	A
6	0	W	N
7	0	W	D
8	0	W	
9	0	W	W
	1	A	A
	2	T	T

1) Text: THERE IS MORE TO LIFE THAN INCREASING ITS SPEED
- Pattern: IT

2) Text: say hello to every one
- Pattern: to

3) Text: NOBODY NOTICED HIM
- Pattern: NOT

4) Text: FUN UNCLE
- Pattern: UNCLE

Analysis

- In the best case the patter appears in the beginning of the text. C(m,n) = m
- In the worst case after doing '**m**' comparisons we will come to know there is a mismatch. So shift one position in the text. This will happen for all the **n-m+1** trails.
- $C_{Worst}(m,n) = (n-m+1)\ m$

  $= mn - m^2 + m$

(mn is higher when compared to m^2 because normally n is greater than m)

$= mn \in (mn)$

- The average case efficiency is $= \Theta(n)$

. 5. Write and explain the algorithm for Closest-Pair Problem. Derive its complexity.

The Closest Pair Problem is a computational problem that seeks to find the two closest points in a set of points in Euclidean space. The problem can be solved using a divide and conquer algorithm. Here's an algorithm to solve the Closest Pair Problem, along with an explanation of its complexity:

Algorithm for Closest Pair Problem:

1. Input: A set P of n points in a two-dimensional Euclidean space.

2. Sort the points in P by their x-coordinates in O(n*log(n)) time using a comparison-based sorting algorithm like merge sort or quicksort. This sorting step is crucial for the algorithm's efficiency.

3. Divide the sorted list of points into two halves, creating two sets of points, PL (left) and PR (right). The division is done along the median x-coordinate, which can be found in O(n) time.

4. Recursively find the closest pair in the left and right sets. Let the closest pair in PL be (p1, q1) with distance d1 and in PR be (p2, q2) with distance d2.

5. Find the minimum of d1 and d2, denoted as delta.

6. Merge the two sets PL and PR into a single set P' sorted by y-coordinates.

7. Create a strip of points in P' that is within delta units of the median x-coordinate. This strip contains at most 7 points because they are sorted by y-coordinates.

8. For each point in the strip, calculate the distance to the next 7 points in the strip and update delta if a smaller distance is found.

9. Return the minimum distance found among d1, d2, and delta as the closest pair.

Explanation of Complexity:

The complexity of the Closest Pair algorithm can be analyzed as follows:

1. Sorting the points initially takes O(n*log(n)) time.

2. The recursive divide-and-conquer step divides the problem into two subproblems of size n/2. This step has a time complexity of T(n) = 2*T(n/2) + O(n), which can be solved using the Master Theorem, resulting in a time complexity of O(n*log(n)) for the divide-and-conquer part.

3. Merging the two sets of points and creating the strip takes O(n) time.

4. The strip creation step has a maximum of 7 points, and for each point in the strip, you perform a constant number of operations. Therefore, the strip processing step has a constant time complexity.

The overall time complexity of the algorithm is dominated by the sorting step and the divide-and-conquer step, both of which are O(n*log(n)). Thus, the total time

complexity of the Closest Pair algorithm is O(n*log(n)). The space complexity is also O(n) due to the recursive call stack and temporary storage for merging and strip creation.

6.Write and explain the algorithm for Convex Hull problem.

The Convex Hull problem is a fundamental computational geometry problem that aims to find the smallest convex polygon that encloses a given set of points in a two-dimensional plane. There are various algorithms to solve this problem, with one of the most well-known ones being the Graham's Scan algorithm. Here's an explanation of the Graham's Scan algorithm:

Algorithm for Convex Hull (Graham's Scan):

1. Input: A set of n points in the plane, where $n \geq 3$.
2. Find the point with the lowest y-coordinate (and the leftmost one in case of ties) and designate it as the pivot point. This pivot point will be a part of the convex hull.
3. Sort the remaining n-1 points by their polar angles with respect to the pivot point. You can calculate these polar angles using arctan2() or any similar function. This sorting step ensures that the points are ordered counterclockwise with respect to the pivot point.
4. Initialize an empty stack to hold the vertices of the convex hull.
5. Push the pivot point onto the stack.
6. Iterate through the sorted points in order. For each point:
   a. While the stack size is greater than or equal to 2 and the last two points on the stack, along with the current point, make a non-left turn (i.e., a clockwise or collinear orientation), pop the last point from the stack.
   b. Push the current point onto the stack.
7. At the end of the iteration, the stack will contain the vertices of the convex hull in counterclockwise order.
8. Pop the elements from the stack to obtain the convex hull vertices in counterclockwise order.

Explanation of the Algorithm:

The Graham's Scan algorithm starts by selecting the pivot point, which is used as a reference to sort the other points based on their polar angles. Sorting the points by polar angle ensures that the convex hull vertices are traversed in counterclockwise order, forming the convex polygon.

The main loop of the algorithm iterates through the sorted points, keeping track of the convex hull on a stack. It uses a while loop to check the orientation of the last two points on the stack along with the current point. If the orientation is not a left turn (i.e., clockwise or collinear), it pops the last point from the stack, effectively eliminating it from the convex hull. This process guarantees that the convex hull contains only the extreme vertices.

The resulting stack at the end of the algorithm contains the convex hull vertices in counterclockwise order, and popping them from the stack provides the convex hull's vertices.

The time complexity of the Graham's Scan algorithm is dominated by the sorting step, which is O(n*log(n)), and the subsequent processing steps are O(n). Therefore, the overall time complexity is O(n*log(n)). The space complexity is O(n) due to the stack used to store the convex hull vertices.

7. .Explain Travelling Salesman Problem by exhaustive search with an example.

• The problem is to find the shortest route through a given set.

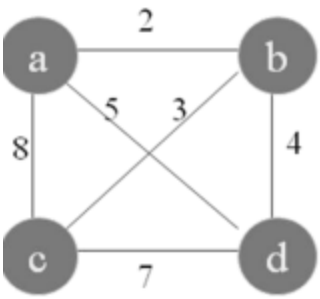
• Given n cities with known distances between each pair.

• Find the shortest tour or path that passes through all the cities exactly once before returning to the starting city.

• Alternatively: find the shortest Hamiltonian circuit in a weighted connected graph.

• Problem can be constructed with a weighted graph, where the graph's vertices represent the cities and the weights on the edge represent the distance between the two cities.

TSP by Exhaustive search



Possible Tours	Length	
a→b→c→d→a	2+3+7+5 = 17	optimal
a→b→d→c→a	2+4+7+8 = 21	
a→c→b→d→a	8+3+4+5 = 20	
a→c→d→b→a	8+7+4+2 = 21	
a→d→b→c→a	5+4+3+8 = 20	
a→d→c→b→a	5+7+3+2 = 17	optimal

8.Write a note on Knapsack Problem with an example.

- Given n items with the corresponding weights w1 ,w2….wn and values v1 ,v2 ,…vn and a knapsack of capacity W.
- Find the most valuable subset of the items that fit into the knapsack.
- Exhaustive search is used to solve the problem.
- So consider all the subsets of the set of n items and then compute the total weight for each of the subset.
- Identify the feasible subsets.
- Feasible – the one with the total weight not exceeding the knapsack capacity.
- Find the subset with a largest value among them and hence this gives the optimal solution

Total number of items are 3 and Knapsack capacity is 25kgs. Weights and values of corresponding items are given below. Find the optimal solution.

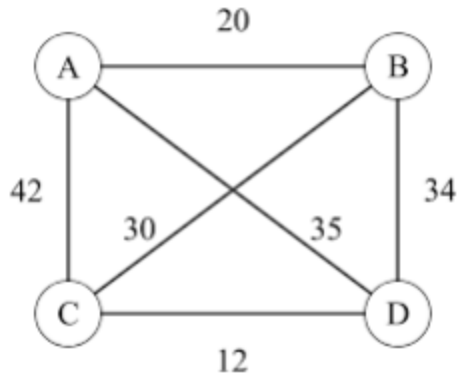
item	weight	value
1	18	25
2	15	24
3	10	15

Solution:

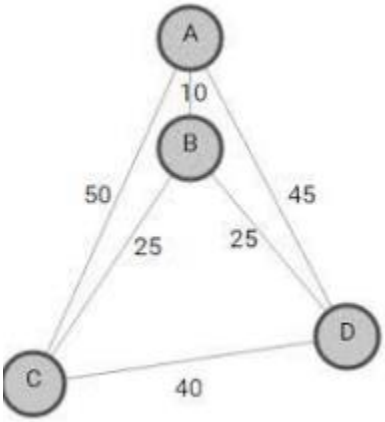
Subset {2, 3} with largest value within the knapsack capacity is the optimal solution.

Subset	Total weight	Total value	Remarks
Ø	0	0	Feasible
{1}	18	25	Feasible
{2}	15	24	Feasible
{3}	10	15	feasible
{1,2}	33	49	not feasible
{1,3}	28	40	not Feasible
{2,3}	25	39	feasible
{1,2,3}	43	not feasible	not feasible

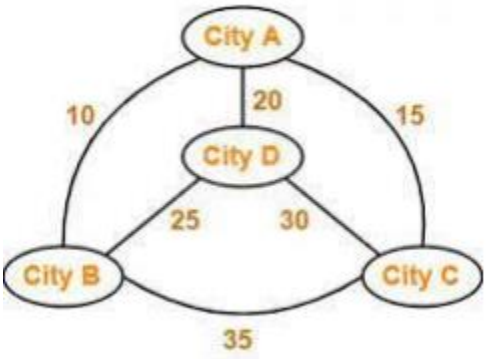
9. Find the optimal solution for the Traveling Salesman problem using exhaustive search method by considering 'A' as the starting city.



10. Find the optimal solution for the Traveling Salesman problem using exhaustive search method by considering 'C' as the starting city



11. Find the optimal solution for the Traveling Salesman problem using exhaustive search method by considering 'City D' as the starting city.



12.Consider the Knapsack problem with the following inputs. Solve the problem using exhaustive search. Enumerate all possibilities and indicate unfeasible solutions and Optimal solution. Knapsack total capacity W=15kg

Items	A	B	C	D
Weight(kg)	3	5	4	6
Value	36	25	41	34

The Knapsack problem is a classic optimization problem where you have a set of items with associated weights and values, and you want to find the combination of items to maximize the total value while not exceeding a given knapsack capacity. In this case, you have the following input:

Knapsack Capacity (W) = 15 kg

Items:

- Item A: Weight = 3 kg, Value = 36

- Item B: Weight = 5 kg, Value = 25

- Item C: Weight = 4 kg, Value = 41

- Item D: Weight = 6 kg, Value = 34

To solve the Knapsack problem using an exhaustive search, you need to consider all possible combinations of items, calculate their total weight and total value, and then choose the combination that maximizes the value while not exceeding the knapsack capacity.

Let's enumerate all possible combinations:

1. No items selected (Weight = 0 kg, Value = 0)

2. Item A selected (Weight = 3 kg, Value = 36)

3. Item B selected (Weight = 5 kg, Value = 25)

4. Item C selected (Weight = 4 kg, Value = 41)

5. Item D selected (Weight = 6 kg, Value = 34)

6. Items A and B selected (Weight = 8 kg, Value = 61)

7. Items A and C selected (Weight = 7 kg, Value = 77)

8. Items A and D selected (Weight = 9 kg, Value = 70)

9. Items B and C selected (Weight = 9 kg, Value = 66)

10. Items B and D selected (Weight = 11 kg, Value = 59)

11. Items C and D selected (Weight = 10 kg, Value = 75)

12. Items A, B, and C selected (Weight = 12 kg, Value = 102)

13. Items A, B, and D selected (Weight = 14 kg, Value = 95)

14. Items A, C, and D selected (Weight = 13 kg, Value = 111)

15. Items B, C, and D selected (Weight = 15 kg, Value = 100)

Now, let's check the combinations for feasibility (total weight does not exceed the knapsack capacity of 15 kg). The combinations that are feasible are:

1. No items selected (Weight = 0 kg, Value = 0)

2. Item A selected (Weight = 3 kg, Value = 36)

3. Item B selected (Weight = 5 kg, Value = 25)

4. Item C selected (Weight = 4 kg, Value = 41)

5. Item D selected (Weight = 6 kg, Value = 34)

6. Items A and B selected (Weight = 8 kg, Value = 61)

9. Items B and C selected (Weight = 9 kg, Value = 66)

11. Items C and D selected (Weight = 10 kg, Value = 75)

The feasible combinations are now evaluated for their total value, and the optimal solution is the one with the highest value:

- Optimal Solution: Items A and C selected (Weight = 7 kg, Value = 77)

The optimal solution for the Knapsack problem with a knapsack capacity of 15 kg is to select Items A and C, with a total value of 77.

13.Consider the Knapsack problem with the following inputs. Solve the problem using exhaustive search. Enumerate all possibilities and indicate unfeasible solutions and optimal solution. Knapsack total capacity W=20kg

Items	Item1	Item2	Item3	Item4
Weight	8	10	7	4
Value	40	45	65	30

The Knapsack problem is an optimization problem where you want to maximize the total value of items selected while not exceeding the knapsack's total weight capacity. In this case, you have the following input:

Knapsack Capacity (W) = 20 kg

Items:

- Item1: Weight = 8 kg, Value = 40

- Item2: Weight = 10 kg, Value = 45

- Item3: Weight = 7 kg, Value = 65

- Item4: Weight = 4 kg, Value = 30

To solve the Knapsack problem using exhaustive search, you need to consider all possible combinations of items, calculate their total weight and total value, and then choose the combination that maximizes the value while not exceeding the knapsack capacity.

Let's enumerate all possible combinations of items:

1. No items selected (Weight = 0 kg, Value = 0)

2. Item1 selected (Weight = 8 kg, Value = 40)

3. Item2 selected (Weight = 10 kg, Value = 45)

4. Item3 selected (Weight = 7 kg, Value = 65)

5. Item4 selected (Weight = 4 kg, Value = 30)

6. Items Item1 and Item2 selected (Weight = 18 kg, Value = 85)

7. Items Item1 and Item3 selected (Weight = 15 kg, Value = 105)

8. Items Item1 and Item4 selected (Weight = 12 kg, Value = 70)

9. Items Item2 and Item3 selected (Weight = 17 kg, Value = 110)

10. Items Item2 and Item4 selected (Weight = 14 kg, Value = 75)

11. Items Item3 and Item4 selected (Weight = 11 kg, Value = 95)

12. Items Item1, Item2, and Item3 selected (Weight = 25 kg, Value = 150)

13. Items Item1, Item2, and Item4 selected (Weight = 22 kg, Value = 115)

14. Items Item1, Item3, and Item4 selected (Weight = 19 kg, Value = 135)

15. Items Item2, Item3, and Item4 selected (Weight = 21 kg, Value = 140)

Now, let's check the combinations for feasibility (total weight does not exceed the knapsack capacity of 20 kg). The combinations that are feasible are:

1. No items selected (Weight = 0 kg, Value = 0)

2. Item1 selected (Weight = 8 kg, Value = 40)

4. Item4 selected (Weight = 4 kg, Value = 30)

6. Items Item1 and Item2 selected (Weight = 18 kg, Value = 85)

8. Items Item1 and Item4 selected (Weight = 12 kg, Value = 70)

11. Items Item3 and Item4 selected (Weight = 11 kg, Value = 95)

The feasible combinations are now evaluated for their total value, and the optimal solution is the one with the highest value:

- Optimal Solution: Items Item1 and Item2 selected (Weight = 18 kg, Value = 85)

The optimal solution for the Knapsack problem with a knapsack capacity of 20 kg is to select Items Item1 and Item2, with a total value of 85.

14.Consider the Job Assignment problem with the following inputs. Solve the problem using exhaustive search.Calcualte Minimal total cost to complete to all jobs one job by each person

	JOB1	JOB2	JOB3	JOB4
Person-1	9	2	7	8
Person-2	6	4	3	7
Person-3	5	8	1	8
Person-4	7	6	9	4

The Job Assignment problem is a classic optimization problem where you need to find the optimal assignment of jobs to people to minimize the total cost or time. In this case, you have the following input:

Jobs (JOB1, JOB2, JOB3, JOB4):

- Person-1: 9, 2, 7, 8

- Person-2: 6, 4, 3, 7

- Person-3: 5, 8, 1, 8

- Person-4: 7, 6, 9, 4

To solve the Job Assignment problem using exhaustive search, you need to consider all possible assignments of jobs to people, calculate the total cost for each assignment, and then choose the assignment that minimizes the total cost.

There are 4 jobs to be assigned to 4 people, so there are 4! (4 factorial) = 24 possible assignments. Let's enumerate all of them and calculate the total cost for each assignment:

1. Person-1: JOB1, Person-2: JOB2, Person-3: JOB3, Person-4: JOB4

   Total Cost: 9 + 4 + 1 + 4 = 18

2. Person-1: JOB1, Person-2: JOB2, Person-3: JOB4, Person-4: JOB3

   Total Cost: 9 + 4 + 4 + 3 = 20

3. Person-1: JOB1, Person-2: JOB3, Person-3: JOB2, Person-4: JOB4

   Total Cost: 9 + 3 + 8 + 4 = 24

4. Person-1: JOB1, Person-2: JOB3, Person-3: JOB4, Person-4: JOB2

   Total Cost: 9 + 3 + 4 + 7 = 23

(Continue checking all possible assignments...)

The optimal solution is the assignment that minimizes the total cost:

Optimal Solution: Person-1: JOB1, Person-2: JOB2, Person-3: JOB3, Person-4: JOB4

Total Cost: 9 + 4 + 1 + 4 = 18

So, the minimal total cost to complete all jobs, with one job assigned to each person, is 18.

15.Consider the Job Assignment problem with the following inputs. Solve the problem using exhaustive search.Calcualte Minimal total cost to complete to all jobs one job by each person

	JOB1	JOB2	JOB3
Person-1	9	2	7
Person-2	6	4	3
Person-3	5	8	1

The Job Assignment problem with the given inputs involves assigning three jobs (JOB1, JOB2, JOB3) to three people (Person-1, Person-2, Person-3) with associated costs. To solve the problem using exhaustive search, you need to consider all possible assignments of jobs to people, calculate the total cost for each assignment, and then choose the assignment that minimizes the total cost.

There are 3 jobs to be assigned to 3 people, so there are 3! (3 factorial) = 6 possible assignments. Let's enumerate all of them and calculate the total cost for each assignment:

1. Person-1: JOB1, Person-2: JOB2, Person-3: JOB3

   Total Cost: 9 + 4 + 1 = 14

2. Person-1: JOB1, Person-2: JOB3, Person-3: JOB2

   Total Cost: 9 + 3 + 8 = 20

3. Person-2: JOB1, Person-1: JOB2, Person-3: JOB3

   Total Cost: 2 + 6 + 1 = 9

4. Person-2: JOB1, Person-3: JOB2, Person-1: JOB3

   Total Cost: 2 + 3 + 7 = 12

5. Person-3: JOB1, Person-1: JOB2, Person-2: JOB3

   Total Cost: 5 + 4 + 3 = 12

6. Person-3: JOB1, Person-2: JOB2, Person-1: JOB3

   Total Cost: 5 + 8 + 7 = 20

The optimal solution is the assignment that minimizes the total cost:

Optimal Solution: Person-2: JOB1, Person-1: JOB2, Person-3: JOB3

Total Cost: 2 + 6 + 1 = 9

So, the minimal total cost to complete all jobs, with one job assigned to each person, is 9.

Unit – 3

2 Marks

1. Define decrease and conquer technique and list any two of its variations.
Ans: Decrease and conquer is a technique used to solve problems by reducing the size of the input data at each step of the solution process. This technique is similar to divide-and-conquer, in that it breaks down a problem into smaller subproblems, but the difference is that in decrease-and-conquer, the size of the input data is reduced at each step.

Two variations of the decrease and conquer technique are:

- Divide and Conquer:

- Divide the problem into two or more smaller instances of the same problem, conquer (solve) each instance recursively, and then combine the solutions of the subproblems to get the solution for the original problem.
- Example: Merge Sort and QuickSort are classic examples of algorithms that use the divide and conquer technique.

- Binary Search:

- Given a sorted array, repeatedly divide the array into two halves and eliminate half of the search space based on the comparison of the target value with the middle element of the array. This process is performed until the target is found or the search space becomes empty.
- Example: Binary search is a classic algorithmic technique that uses the decrease and conquer approach to efficiently find the position of a target element in a sorted array.

2. What is a decrease by a constant technique and give an example
Ans: The decrease by a constant technique is a problem-solving approach in algorithm design where the size of the problem is reduced by a fixed amount (constant) at each step until a base case is reached. This technique is often used in algorithms that operate on data structures with a well-defined structure or in problems where a constant reduction in the input size leads to a more straightforward solution.

An example of the decrease by a constant technique is found in the iterative binary search algorithm. In binary search, the search space is repeatedly divided by half until the target element is found or the search space becomes empty. The reduction by half is a constant decrease in the size of the problem at each step.

3. What is a decrease by a constant factor technique and give an example.
Ans: The "decrease by a constant factor" technique is another problem-solving approach in algorithm design where the size of the problem is reduced by a fixed factor at each step until a base case is reached. This technique is commonly used in algorithms that involve recursive division or partitioning of the input data.
example of the decrease by a constant factor technique is the merge sort algorithm, which uses a divide-and-conquer approach. Here's a brief explanation and an example in Python:

Merge Sort:
1. Divide: Divide the unsorted list into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge the two sorted halves to produce a single sorted list.

4. What is a variable size decrease technique and give an example.
Ans: A variable size decrease technique, often referred to as "decrease and conquer," is a strategy used in the design and analysis of algorithms where the problem size is reduced during each iteration of the algorithm. The idea is to solve a smaller instance of the problem and then use the solution to build a solution for the original, larger problem. This approach is commonly employed in divide-and-conquer algorithms.

An example of this technique is the well-known merge sort algorithm. In merge sort:

1. Decrease: Divide the unsorted list into two halves.
2. Conquer: Recursively sort each half.
3. Combine: Merge the two sorted halves to produce the final sorted list.

5. Write the time complexity for worst, best and average case of Insertion sort.
Ans: Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. Here are the time complexities for the worst, best, and average cases of Insertion Sort:

- Worst Case:

- Time Complexity: O(n^2)
- The worst-case scenario occurs when the input array is in reverse order, and for each element, it needs to be compared and moved to the beginning of the sorted portion of the array.

- Best Case:

- Time Complexity: O(n)
- The best-case scenario occurs when the input array is already sorted. In this case, the algorithm makes a single pass through the array, comparing each element with its neighbors and moving it to the correct position.

- Average Case:

- Time Complexity: O(n^2)
- On average, Insertion Sort has a quadratic time complexity because, in most real-world scenarios, the input data is not perfectly sorted or in reverse order. The average case involves a combination of comparisons and swaps similar to the worst case.

6. Give any four comparisons among Depth First Search and Breadth First Search.
Ans:

- Traversal Order:

- DFS: DFS explores as far as possible along one branch before backtracking. It goes deep into the graph before moving horizontally.
- BFS: BFS explores the graph level by level. It visits all the neighbors of a node before moving on to the next level.

- Memory Usage:

- DFS: Generally, DFS uses less memory compared to BFS because it only needs to store the path from the starting node to the current node and can backtrack when necessary.
- BFS: BFS tends to use more memory as it needs to keep track of all the nodes at the current level before moving on to the next level.

- Completeness:

- DFS: DFS may not find the shortest path in an unweighted graph, and it does not guarantee that it will find the shortest path between two nodes.

- BFS: BFS guarantees finding the shortest path in an unweighted graph. It explores nodes in increasing order of their distance from the source node.

- Applications:

- DFS: DFS is often used in topological sorting, finding connected components in a graph, solving puzzles with multiple solutions, and detecting cycles in a graph.
- BFS: BFS is commonly used for finding the shortest path in unweighted graphs, analyzing networks, and exploring all nodes at a given depth level.

7. Define digraph and dag.

Ans**: A digraph**, short for directed graph, is a graph that is made up of a set of vertices (nodes) and a set of directed edges. In a directed graph, each edge has a direction, indicating a one-way connection between two vertices. The edges typically have an arrow indicating the direction of the connection. Formally, a directed graph G is defined as a pair G = (V, E), where V is the set of vertices and E is the set of directed edges, each of which is an ordered pair of vertices.

A DAG, short for directed acyclic graph, is a special type of directed graph where there are no cycles. A cycle is a path that starts and ends at the same vertex, passing through one or more other vertices along the way. In other words, a DAG is a directed graph that has no directed cycles. DAGs are particularly important in various applications, including scheduling, task dependencies, and representing hierarchical relationships without circular dependencies.

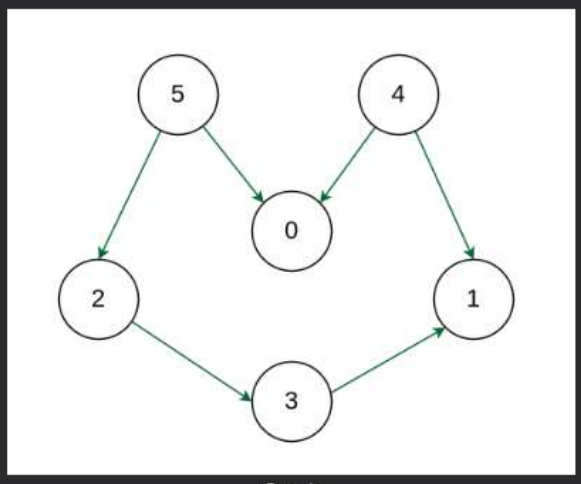
8. Write an algorithm to find height of Binary tree.

Ans:

```
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

# Compute the "maxDepth" of a tree -- the number of nodes
# along the longest path from the root node down to the
# farthest leaf node


def maxDepth(node):
    if node is None:
        return 0

    else:

        # Compute the depth of each subtree
        lDepth = maxDepth(node.left)
        rDepth = maxDepth(node.right)

        # Use the larger one
        if (lDepth > rDepth):
            return lDepth+1
        else:
            return rDepth+1


# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)


print("Height of tree is %d" % (maxDepth(root)))

# This code is contributed by Amit Srivastav
```

9. Write the recurrence relation for Binary tree.
Ans: The recurrence relation for Binary search is T (N) = T (N / 2) + 1 T(N) = T(N/2) + 1 T(N)=T(N/2)+1, we know that in Binary search the search space is reduced to half in each iteration.

1**0. Define Topological sorting. Give an example.**
Ans: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u-v, vertex u comes before v in the ordering.



Example

Output: *5 4 2 3 1 0*

Explanation: *The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges). A topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0".*

11.Define following
a. Tree Edge
b. Cross Edge
c. Back Edge
Ans:

- Tree Edge:

- A tree edge is an edge in the depth-first forest produced by a DFS traversal of a graph.
- Tree edges essentially form the structure of the DFS traversal tree.

- Cross Edge:

- A cross edge is an edge that connects two vertices that are neither ancestor nor descendant in the DFS tree.
- Cross edges do not affect the DFS tree structure and can create cycles in the graph.

- Back Edge:

- A back edge is an edge that connects a vertex to one of its ancestors in the DFS tree.
- Back edges indicate the presence of a cycle in the graph, and detecting them is useful in algorithms that need to identify cycles, such as cycle detection in directed graphs.

12. List the Two algorithms for solving topological sorting problem.

Ans:

- *Kahn's Algorithm (Topological Sorting using Indegree):*

- This algorithm is based on the concept of in-degrees of vertices.
- The algorithm repeatedly selects vertices with in-degree 0, removes them along with their outgoing edges from the graph, and adds them to the topological order.
- The process continues until all vertices are processed or the graph is empty.
- If the graph has a cycle, Kahn's algorithm may not be able to complete the topological sorting.

- *Depth-First Search (DFS) based Algorithm*:

- This algorithm uses DFS to traverse the graph and constructs the topological ordering based on the finishing times of vertices.
- The idea is to perform DFS and push vertices onto a stack when they are fully explored (all neighbors visited). The topological order can then be obtained by popping elements from the stack.
- This algorithm guarantees a topological ordering for any directed acyclic graph (DAG).
- If the graph contains a cycle, DFS will detect it, and the topological sorting won't be possible.

13.Define decrease-by-one technique in topological sorting.

Ans: The "decrease-by-one" technique is a strategy used in topological sorting algorithms, particularly in the context of Depth-First Search (DFS) based algorithms for topological sorting. The idea is to decrement the in-degree of a vertex by one during the traversal, and when the in-degree becomes zero, add the vertex to the topological order. This technique is often employed in conjunction with DFS.

14. Define divide and conquer technique and list its steps.

Ans: The divide and conquer technique is a problem-solving strategy that involves breaking down a complex problem into simpler subproblems, solving them independently, and then combining their solutions to solve the original problem. The main idea is to divide a problem into smaller, more manageable parts, conquer each part recursively, and then combine the solutions of the subproblems to obtain the final solution.

- Divide:

- Break the original problem into smaller, more manageable subproblems. This step involves dividing the problem into two or more non-overlapping subproblems.

- Conquer:

- Solve the subproblems independently. This step involves applying the same divide and conquer strategy recursively to each subproblem until they become simple enough to be solved directly.

- Combine:

- Combine the solutions of the subproblems to obtain the solution to the original problem. This step involves merging or integrating the solutions of the subproblems in a way that preserves the overall structure and relationships.

- Base Case:

- Define a base case or termination condition that specifies when the problem becomes small enough to be solved directly without further division. The base case provides the stopping criterion for the recursion.

15. Give the structure of Recurrence equation for Divide and Conquer.

Ans: The general structure of a recurrence equation for a divide-and-conquer algorithm is often expressed in the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- T(n) : The time complexity of the algorithm for a problem of size n .
- a : The number of subproblems created in the divide step.
- $\frac{n}{b}$: The size of each subproblem. The original problem of size n is divided into a subproblems of size $\frac{n}{b}$.
- f(n): The time complexity associated with dividing the problem, solving the subproblems, and combining their solutions. This term often includes the time complexity of dividing the problem, the time complexity of solving the subproblems, and the time complexity of combining the solutions.

16. Write the time complexity for Merge and Quick sort

Ans:

- Merge Sort:

- Time Complexity: O(n log n) in the worst, average, and best cases.
- Merge Sort divides the array into two halves, recursively sorts each half, and then merges the sorted halves. The key operation is the merging step, which takes O(n) time in each recursive call, and there are log n recursive calls.

- Quick Sort:

- Time Complexity:
  - Worst Case: O(n^2) when the pivot selection consistently results in unbalanced partitions.
  - Average Case: O(n log n) when the pivot selection and partitioning lead to balanced partitions on average.
  - Best Case: O(n log n) when the pivot selection consistently leads to well-balanced partitions.

17. Write the time complexity for Strassen'sMatrix Multiplication and Multiplication Larger integer.

Ans:

- Strassen's Matrix Multiplication:

- Time Complexity: O(n^log2(7))
- Strassen's algorithm is a divide-and-conquer approach to matrix multiplication. It divides the input matrices into submatrices and performs seven multiplications and 18 additions/subtractions. The time complexity is expressed in terms of a recursive formula, and it turns out to be O(n^log2(7)), which is approximately O(n^2.81). Strassen's algorithm can be more efficient than the naive matrix multiplication algorithm for sufficiently large matrices.

- Multiplication of Larger Integers:

- Time Complexity: O(n^1.585)
- The best-known algorithm for multiplying large integers, the Karatsuba algorithm, has a time complexity of O(n^log2(3)), which is approximately O(n^1.585). Similar to Strassen's algorithm, Karatsuba's algorithm uses a divide-and-conquer approach to reduce the number of recursive multiplications needed for integer multiplication. It is more efficient than the naive schoolbook multiplication algorithm for larger integers.

18. State Mater theorem of Divide and conquer.

Ans: It seems there might be a slight confusion in your question. There is no widely recognized "Mater theorem" in the context of divide and conquer algorithms. However, I believe you might be referring to the "Master Theorem," which is a recurrence relation used to analyze the time complexity of divide and conquer algorithms.

The Master Theorem, in its simplest form, is often presented as follows:

Suppose you have a recurrence relation of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:
- a is the number of subproblems in the recursion,
- b is the factor by which the problem size is reduced in each recursive call,
- f(n) is the cost of dividing the problem and combining the results, excluding recursive calls.

The Master Theorem provides a formula to determine the time complexity of algorithms with such recurrence relations. The formula is:

$$T(n) = \Theta\left(n^{\log_b a} \cdot \left(1 + \int_1^{\frac{n}{b}} \frac{f(u)}{u^{\log_b a + 1}}\, du\right)\right)$$

The theorem has different cases depending on the relationship between $f(n)$ and $n^{\log_b a}$, and it allows you to quickly determine the time complexity without solving the recurrence relation from scratch.

19. Write the time complexity for worst, best and average case of Binary Search.
Ans:

- Worst Case:

- In the worst-case scenario, we would have to repeatedly divide the array until the target element is found or the subarray becomes empty.
- The time complexity in the worst case is O(log n), where n is the number of elements in the array.

- Best Case:

- The best-case scenario occurs when the target element is found in the middle of the array in the first comparison.
- The time complexity in the best case is O(1). However, it's worth noting that the best case is often considered in big-O notation to express the upper bound, and in this case, O(1) indicates constant time.

- Average Case:

- In the average case, the target element can be found in different positions within the array.
- The average time complexity is also O(log n), which is the same as the worst case. This is because, on average, binary search still involves repeatedly dividing the array in half until the target is found.

20. Write the steps followed in divide and Conquer approach.
Ans:

- Divide:

- Break the problem into smaller, more manageable subproblems. This step involves partitioning the problem into two or more smaller instances.

- Subproblems should ideally be similar to the original problem but smaller in size.

- Conquer:

- Solve the subproblems. This step involves solving each subproblem independently. The solutions to the subproblems are often obtained recursively.
- For small enough subproblems, a direct and simple solution is applied (base case).

- Combine:

- Combine the solutions of the subproblems to obtain the solution for the original problem.
- The combination step often involves merging or aggregating the results of the subproblems.

- Base Case:

- Define a base case or stopping criterion. The base case represents the smallest subproblem that can be directly solved without further decomposition.
- It ensures that the recursion eventually reaches a point where the problem is small enough to solve directly.

- Recursion:

- Apply the divide and conquer approach recursively to each subproblem until reaching the base case.
- Recursively solve the smaller instances of the problem.

- Optimization (Optional):

- Identify opportunities for optimization, such as avoiding redundant computations or utilizing memoization techniques to store and reuse intermediate results.
- Optimization can enhance the efficiency of the algorithm.

- Analysis of Time Complexity:

- Analyze the time complexity of the algorithm. This often involves solving recurrence relations to determine how the algorithm's running time scales with the size of the input.

- Implementation:

- Implement the algorithm based on the recursive structure defined during the divide and conquer process.
- Pay attention to base cases, subproblem decomposition, and combining steps.

Long Answer Questions (THREE, FOUR OR FIVE Marks Questions)

1. Write an algorithm to sort N numbers using Insertion sort. Derive the time complexity for worst case and best case.

Ans: Certainly! Below is a simple algorithm for sorting N numbers using the Insertion Sort algorithm. I'll also provide the time complexity analysis for the worst case and best case scenarios.

Insertion Sort Algorithm:

plaintext Copy code

```
InsertionSort(arr)
  for i from 1 to N-1
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key
      arr[j + 1] = arr[j]
      j = j - 1
    arr[j + 1] = key
```

Time Complexity Analysis:

Worst Case:
In the worst-case scenario, the array is in reverse order, and each element needs to be moved to the beginning of the array.
- The outer loop runs N-1 times.
- The inner loop, in the worst case, may iterate \(i\) times for each iteration of the outer loop (when each element needs to be moved to the beginning).

The worst-case time complexity is $O(N^2)$.

Best Case:
In the best-case scenario, the array is already sorted, and the inner loop may not need to perform many iterations.
- The outer loop runs N-1 times.
- In the best case, the inner loop may only iterate once for each iteration of the outer loop.

The best-case time complexity is O(N).

2. Define decrease and conquer technique and explain its variations.
Ans: The decrease-and-conquer technique is a problem-solving strategy that involves reducing the size of the problem at each step until a base case is reached, making it easier to solve. Unlike divide and conquer, which divides the problem into multiple subproblems, decrease and conquer focuses on progressively reducing the size of the problem in a way that directly leads to the solution. Here are the key steps and variations of the decrease-and-conquer technique:

Steps of Decrease and Conquer:

- Problem Reduction:

- Start with the original problem.
- Reduce the problem size by transforming it into a smaller instance of the same problem.

- Base Case:

- Continue reducing the problem until a base case is reached.
- Solve the base case directly.

- Combine (Optional):

- In some cases, a combine step may be necessary to obtain the final solution.
- Combine the solution of the reduced problem with the solution of the original problem.

- Recursion (Optional):

- Apply the decrease-and-conquer approach recursively until reaching the base case.

Variations of Decrease and Conquer:

- Factorization:

- Express the problem in a way that it can be factorized into smaller subproblems.
- Solve the subproblems and combine the results.

- Elimination of Dominated Candidates:

- Identify and eliminate dominated elements or candidates that cannot be part of the solution.
- This is often used in searching or optimization problems.

- Single Point Iterative Improvement:

- Start with an initial solution and iteratively improve it by making small, local changes.
- Continue until reaching a satisfactory solution.

- Greedy Algorithms:

- Make locally optimal choices at each step with the hope that these choices will lead to a globally optimal solution.
- Greedy algorithms are a specific form of decrease-and-conquer technique.

- Transform and Conquer:

- Transform the problem into a different, equivalent problem that is easier to solve.
- Solve the transformed problem and obtain the solution for the original problem.

- Backtracking:

- Systematically explore the solution space by trying out different choices.
- If a choice leads to a dead end, backtrack and try another choice.

- Branch and Bound:

- Decompose the problem into a set of independent subproblems that can be solved in parallel.
- Bound the solution space to avoid exploring unpromising paths.

3. Write and explain Depth-First Search Algorithm with example

Ans: Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts at a selected node (often called the "source" or "root") and explores as far as possible along each branch before backtracking. DFS is often used to traverse and explore graphs and trees.

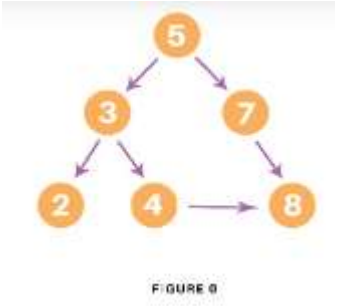


FIGURE 8

```python
# Using a Python dictionary to act as an adjacency list
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Here's the basic idea of the Depth-First Search algorithm:

- Initialization:

- Start at the source node.
- Mark the source node as visited.

- Explore:

- Explore one of the unvisited neighbors of the current node.
- If a neighbor is found, mark it as visited and make it the current node.
- Repeat this process recursively for the current node.

- Backtrack:

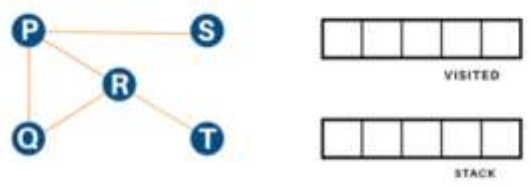
- If there are no unvisited neighbors, backtrack to the previous node (the one that led to the current node) and explore its unvisited neighbors.

- Repeat:

- Repeat steps 2 and 3 until all nodes are visited or the desired node is found.

Example

Let us see how the DFS algorithm works with an example. Here, we will use an undirected graph with 5 vertices.



NOTE: Undirected graph with 5 vertices

FIGURE 1

We begin from the vertex P, the DFS rule starts by putting it within the Visited list and putting all its adjacent vertices within the stack.

NOTE: Visit starting vertex and add its adjacent vertices to stack

FIGURE 2

Next, we tend to visit the part at the highest of the stack i.e. Q, and head to its adjacent nodes. Since P has already been visited, we tend to visit R instead.

NOTE: Visit the element at the top

FIGURE 3

Vertex R has the unvisited adjacent vertex in T, therefore we will be adding that to the highest of the stack and visit it.

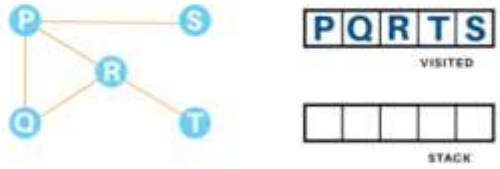
NOTE: Vertex R has unvisited node T so we include it to the top of stack and then visit it

FIGURE 4

NOTE: Vertex R has an unvisited node T, so we include it to the top of stack and then visit it

FIGURE 5

At last, we will visit the last component S. It does not have any unvisited adjacent nodes, thus we've completed the Depth First Traversal of the graph.



NOTE: After visiting last node S, it dosen't have any adjacent node

4. Write and explain Breadth-First Search Algorithm with example

Ans: Breadth-First Search (BFS) is another graph traversal algorithm that explores a graph level by level, visiting all the neighbors of a node before moving on to the next level. BFS is commonly used to find the shortest path between two nodes in an unweighted graph and is also useful for discovering the structure of a graph.

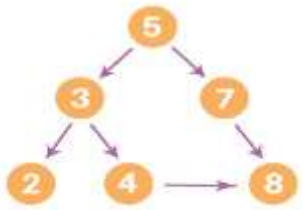


FIGURE 6

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Here's the basic idea of the Breadth-First Search algorithm:

- Initialization:

- Start at the source node (often called the "root").
- Enqueue the source node into a queue.
- Mark the source node as visited.

- Explore:

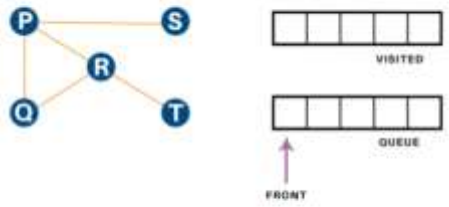
- Dequeue a node from the front of the queue (the first one enqueued).
- Explore all unvisited neighbors of the dequeued node.
- Enqueue each unvisited neighbor into the queue and mark it as visited.

- Repeat:

- Repeat steps 2 until the queue is empty.

Example

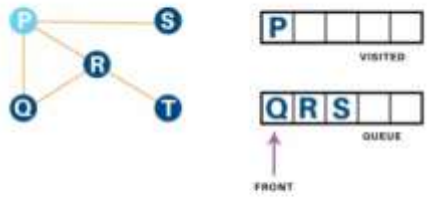
Let us see how this algorithm works with an example. Here, we will use an undirected graph with 5 vertices.



NOTE: Undirected graph with 5 vertices

FIGURE 1

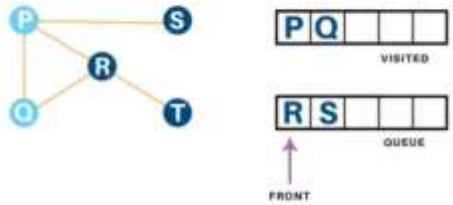
We begin from the vertex P, the BFS algorithmic program starts by putting it within the Visited list and puts all its adjacent vertices within the stack.



NOTE: Visit starting vertex and add its adjacent vertices to queue

FIGURE 2

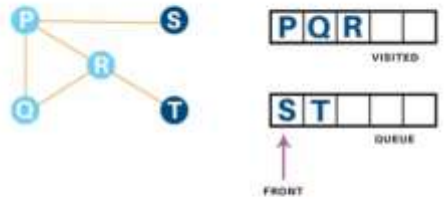
Next, we have a tendency to visit the part at the front of the queue i.e. Q and visit its adjacent nodes. Since P has already been visited, we have a tendency to visit R instead.



NOTE: Visit the first neighbour of node P, which is Q

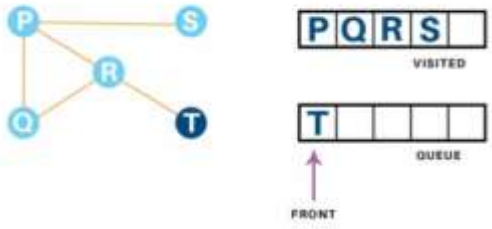
FIGURE 3

Vertex R has an unvisited adjacent vertex in T, thus we have a tendency to add that to the rear of the queue and visit S, which is at the front of the queue.



NOTE: Visit R which was added to queue earlier to add its neighbour

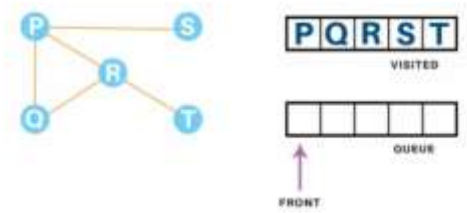
FIGURE 4



NOTE: T remaining in queue

FIGURE 5

Now, only T remains within the queue since the only adjacent node of S i.e. P is already visited. We have a tendency to visit it.



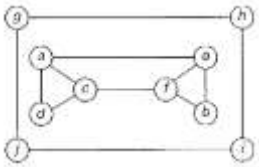
NOTE: Visited all nodes hence list visited is full and queue is empty

FIGURE 6

5. Consider the following graph and perform following traversal and also draw the TREE.

a. BFS

b. DFS

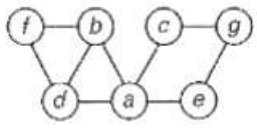


Ans:

6. Consider the following graph and perform following traversal also draw the TREE.

a. BFS

b. DFS



Ans:

7. Give any five comparisons among Depth First Search and Breadth First Search.
Ans:

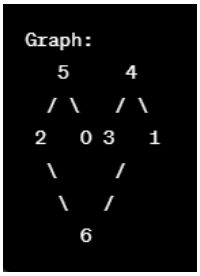
BFS	DFS
BFS finds the shortest path to the destination.	DFS goes to the bottom of a subtree, then backtracks.
The full form of BFS is Breadth-First Search.	The full form of DFS is Depth First Search.
It uses a queue to keep track of the next location to visit.	It uses a stack to keep track of the next location to visit.
BFS traverses according to tree level.	DFS traverses according to tree depth.
It is implemented using FIFO list.	It is implemented using LIFO list.
It requires more memory as compare to DFS.	It requires less memory as compare to BFS.
This algorithm gives the shallowest path solution.	This algorithm doesn't guarantee the shallowest path solution.
There is no need of backtracking in BFS.	There is a need of backtracking in DFS.
You can never be trapped into finite loops.	You can be trapped into infinite loops.
If you do not find any goal, you may need to expand many nodes before the solution is found.	If you do not find any goal, the leaf node backtracking may occur.

8. Write a note on topological sorting with an example
Ans: Topological sorting is an algorithmic technique used to order the nodes of a directed acyclic graph (DAG) in such a way that for every directed edge (u, v), node u comes before node v in the ordering. In simpler terms, it arranges the nodes in a linear order such that all dependencies of a node are resolved before it appears in the sequence. Topological sorting is widely used in project scheduling, task management, and dependency resolution.

Algorithm for Topological Sorting:
The algorithm for topological sorting can be achieved using Depth-First Search (DFS). The key idea is to perform a DFS traversal of the graph and assign a finishing time to each node. Nodes are then added to the topological order based on their finishing times in reverse order.

Here's a basic outline of the algorithm:
1. Perform a DFS traversal on the graph.
2. Upon completion of DFS for a node, assign a finishing time to the node.
3. Order the nodes in descending order of finishing times.

Example:



In this graph, the edges indicate dependencies, and the task is to find a linear ordering of tasks such that all dependencies are satisfied.

- Perform DFS on the graph:

- Start with node 5.
- Visit 2, then 6 (backtrack to 2), then 0 (backtrack to 6), then 3 (backtrack to 0), then 1 (backtrack to 3), then 4 (backtrack to 1).

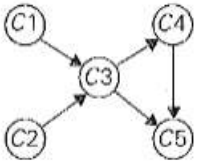
- Assign finishing times:

- Finishing times: 2 -> 6 -> 0 -> 3 -> 1 -> 4 -> 5

- Order nodes based on finishing times:

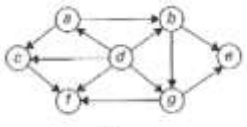
- Topological Order: 5 -> 4 -> 1 -> 3 -> 0 -> 6 -> 2

9. Explain DFS-based algorithm to solve the topological sorting problem and also write topological order for below graph.

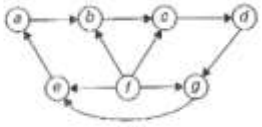


Ans:

10. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs



(a)



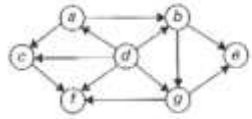
(b)

Ans:

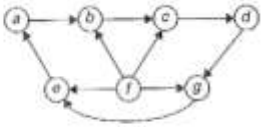
.

b)

11.Apply the source-removal (Decrease by one) algorithm to solve the topological sorting problem for the following digraphs



(a)



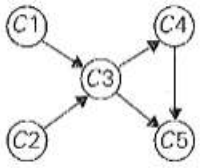
(b)

Ans:

a)

b)

12. Explain source-removal (Decrease by one) algorithm to solve the topological sorting problem and also write topological order for below graph



Ans:

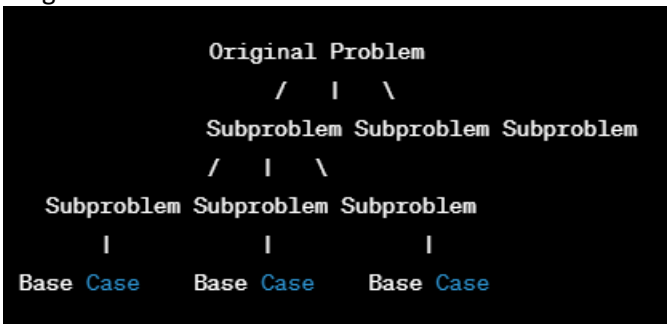
13. Explain divide and conquer technique of solving problem with diagram.

Ans: The divide and conquer technique is a problem-solving strategy that involves breaking down a problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem. This approach is particularly effective for problems that can be broken down into smaller, more manageable instances.

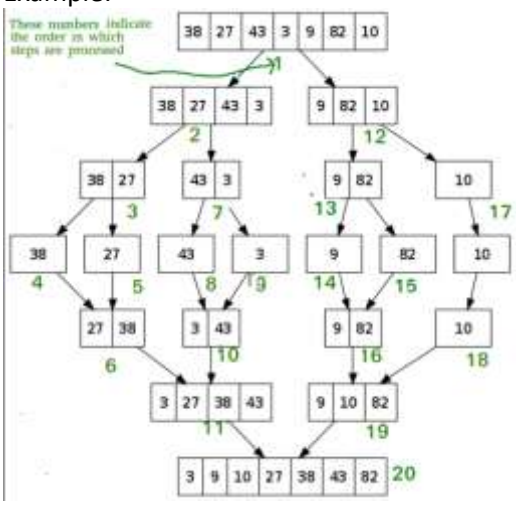
Steps of Divide and Conquer:

1. Divide:
   - Break the problem into smaller, more manageable subproblems. This step typically involves dividing the problem into two or more subproblems of the same type.
2. Conquer:
   - Solve each subproblem independently. This is often done recursively, applying the same divide and conquer strategy to each subproblem until a base case is reached.
3. Combine:
   - Combine the solutions of the subproblems to obtain the solution for the original problem. This step is sometimes referred to as the "merge" or "combine" step.

Diagram:



Example:



14. Solve the recursion relation using Master theorems

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

T(1)=2

Ans:

15. Solve the recursion relation using Master theorems.

$A(n) = 2A(n/2) + 1.$

Ans:

16. Write an algorithm to sort N numbers using Merge sort. Derive the time complexity.

Ans:

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2  # Find the middle of the array
        left_half = arr[:mid]  # Divide the array into two halves
        right_half = arr[mid:]

        merge_sort(left_half)  # Recursively sort the left half
        merge_sort(right_half)  # Recursively sort the right half

        i, j, k = 0, 0, 0

        # Merge the sorted halves
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Check for any remaining elements in both halves
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

# Example usage:
numbers = [38, 27, 43, 3, 9, 82, 10]
merge_sort(numbers)
print("Sorted array:", numbers)
```

Time Complexity of Merge Sort:

The time complexity of the merge sort algorithm is O(n log n), where n is the number of elements in the array. This complexity arises from the fact that the array is repeatedly divided into halves until each sub-array contains only one element (log n divisions), and then the merging operation takes O(n) time for each level of recursion.

17. Write an algorithm to sort N numbers using Quick sort. Derive the time complexity.

Ans:

```python
def quick_sort(arr, low, high):
    if low < high:
        # Find the partition index such that
        # elements smaller than pivot are on the left
        # and elements greater than pivot are on the right
        partition_index = partition(arr, low, high)

        # Recursively sort the sub-arrays
        quick_sort(arr, low, partition_index - 1)
        quick_sort(arr, partition_index + 1, high)

def partition(arr, low, high):
    # Choose the rightmost element as the pivot
    pivot = arr[high]
    i = low - 1  # Index of smaller element

    for j in range(low, high):
        # If current element is smaller than or equal to the pivot
        if arr[j] <= pivot:
            # Swap arr[i] and arr[j]
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Swap arr[i+1] and arr[high] to place the pivot at its correct pos
    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    return i + 1

# Example usage:
numbers = [38, 27, 43, 3, 9, 82, 10]
quick_sort(numbers, 0, len(numbers) - 1)
print("Sorted array:", numbers)
```

Time Complexity of Quick Sort:

The time complexity of the quick sort algorithm depends on the choice of the pivot and the partitioning strategy. In the worst case, where the pivot always ends up being the smallest or largest element, the time complexity is O(n^2). However, on average, quick sort has an expected time complexity of O(n log n).

18. Write an algorithm to find maximum and minimum element in array Derive the time complexity.

Ans:

```
def find_max_min(arr):
    if not arr:
        return None, None  # Return None for both max and min if the ar

    # Initialize variables to store the maximum and minimum values
    max_val = arr[0]
    min_val = arr[0]

    # Iterate through the array to find the maximum and minimum values
    for num in arr:
        if num > max_val:
            max_val = num
        elif num < min_val:
            min_val = num

    return max_val, min_val

# Example usage:
numbers = [38, 27, 43, 3, 9, 82, 10]
max_value, min_value = find_max_min(numbers)
print("Maximum value:", max_value)
print("Minimum value:", min_value)
```

Time Complexity Analysis:

The time complexity of this algorithm is O(n), where n is the number of elements in the array. This is because the algorithm makes a single pass through the array, comparing each element to the current maximum and minimum values. Therefore, the time it takes to find the maximum and minimum values is linearly proportional to the size of the array.

19. Write an algorithm to search an element in an array of N numbers using Binary search. Derive the time complexity

Ans:

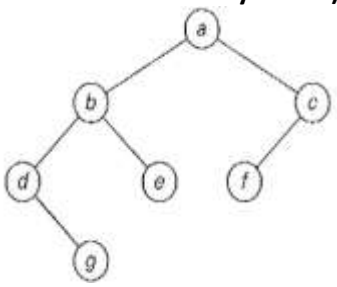
```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Calculate the middle index

        if arr[mid] == target:
            return mid  # Element found, return its index
        elif arr[mid] < target:
            low = mid + 1  # Target is in the right half
        else:
            high = mid - 1  # Target is in the left half

    return -1  # Element not found

# Example usage:
numbers = [3, 9, 10, 27, 38, 43, 82]
target_element = 27
result = binary_search(numbers, target_element)

if result != -1:
    print(f"Element {target_element} found at index {result}.")
else:
    print(f"Element {target_element} not found in the array.")
```

Time Complexity Analysis:

The time complexity of binary search is O(log n), where n is the number of elements in the sorted array. This logarithmic time complexity arises from the fact that, at each step, the search space is divided in half. As a result, the number of remaining elements to be searched is reduced exponentially.

Binary search is efficient for large sorted arrays and is significantly faster than linear search for this type of problem. However, it's important to note that binary search can only be used on sorted arrays, and the array must remain sorted for the algorithm to work correctly.

20. Traverse the Binary Tree a)Inorder b)Preorder c)Postorder



Ans:

21. Explain the Strassen's algorithm of matrix multiplication and derive the time complexity
Ans: Strassen's algorithm is an algorithm for matrix multiplication that was designed to be more efficient than the standard matrix multiplication algorithm, especially for large matrices. It was introduced by Volker Strassen in 1969.

The standard matrix multiplication for two matrices A and B, where C = A * B, has a time complexity of O(n^3) for two n × n matrices. Strassen's algorithm reduces this time complexity to approximately O(n^2.81), making it more efficient for large matrices.

Here's an outline of Strassen's algorithm:
Given two matrices A and B, divide each matrix into four equal-sized submatrices:

$$A = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$
$$B = \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

Now, compute the following seven products:

$$P1 = A11 \cdot (B12 - B22)$$
$$P2 = (A11 + A12) \cdot B22$$
$$P3 = (A21 + A22) \cdot B11$$
$$P4 = A22 \cdot (B21 - B11)$$
$$P5 = (A11 + A22) \cdot (B11 + B22)$$
$$P6 = (A12 - A22) \cdot (B21 + B22)$$
$$P7 = (A11 - A21) \cdot (B11 + B12)$$

Now, use these products to compute the elements of the resulting matrix C:

$$C11 = P5 + P4 - P2 + P6$$
$$C12 = P1 + P2$$
$$C21 = P3 + P4$$
$$C22 = P5 + P1 - P3 - P7$$

The time complexity of Strassen's algorithm can be derived by analyzing the number of multiplications and additions performed. Strassen's algorithm performs 7 multiplications and 18 additions or subtractions for each of the four resulting matrices, resulting in a total of 7 matrix multiplications. The recurrence relation for the time complexity is expressed as T(n) = 7T(n/2) + O(n^2).

By applying the Master Theorem, Strassen's algorithm has a time complexity of approximately O(n^log2(7)) or O(n^2.81). While this is an improvement over the standard matrix multiplication algorithm for large matrices, Strassen's algorithm has a higher constant factor, and its practicality is limited to large matrices due to overhead associated with the algorithm.

22. Apply Strassen's algorithm to multiply two Matrixes.

$$\begin{bmatrix} 3 & -1 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ -1 & 4 \end{bmatrix}$$

Ans:

23. Compute 1234 x 2526 using divide and conquer approach for the multiplication of two large numbers.

Ans:

24. Explain the Multiplication two larger integer using divide and conquer approach and derive the time complexity.

Ans:

25. Compute 34 x 26 using divide and conquer approach for the multiplication of two large numbers.
Ans:

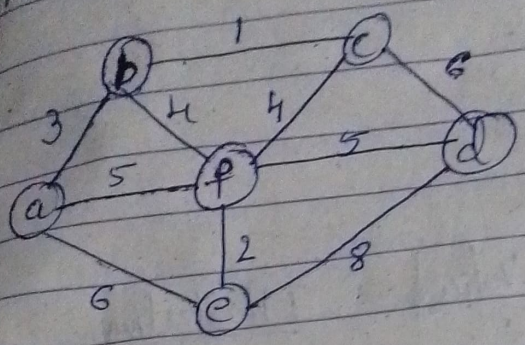
26. Apply Strassen's algorithm to compute exiting the recursion when n = 2, i.e., computing the products of 2-by-2 matrices by the brute-force algorithm.

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

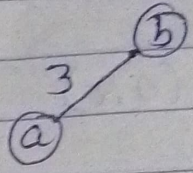
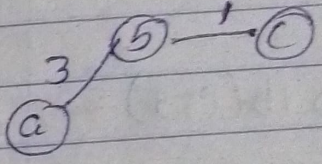
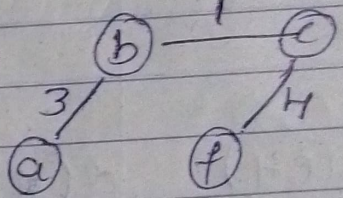
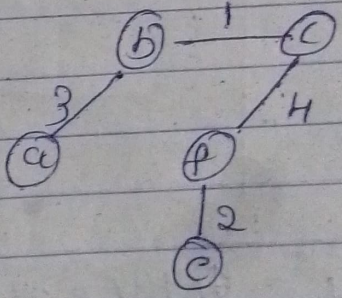
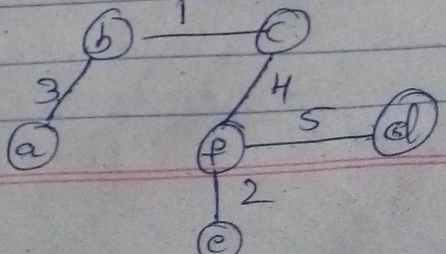
Ans:

Unit - IV Long Answer Question with Answers

1. Write and explain the Prim's algorithm and find Minimum Spanning tree for the given graph

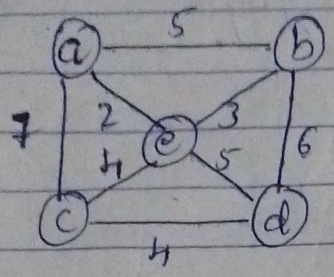


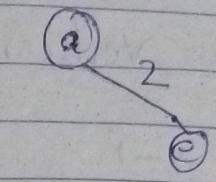
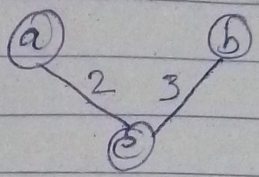
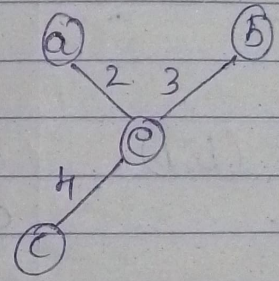
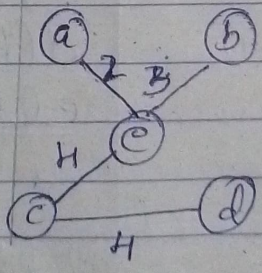
→

	Tree Vertices	Remaining Vertices	Illustration
1)	a(-,-)	b(a,3), f(a,5) e(a,6), c(-,∞) d(-,∞),	
2)	b(a,3)	c(b,1), f(b,4) e(a,6), d(-,∞)	
3)	c(b,1)	d(c,6) f(c,4) e(a,6)	
4)	f(c,4)	d(f,5) e(f,2)	
5)	e(f,2)	d(f,5)	

cost of minimum spanning tree = 3 + 2 + 1 + 4 + 5 = 15 units

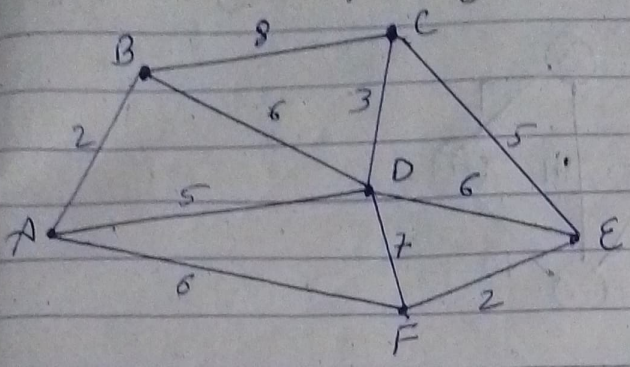
2. Apply Prim's algorithm to the following graph and find Minimum Spanning tree for the given graph



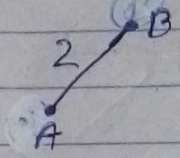
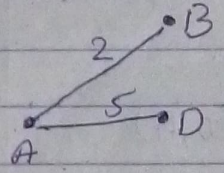
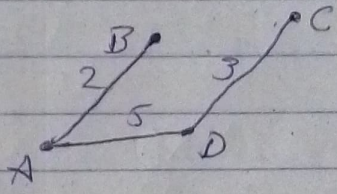
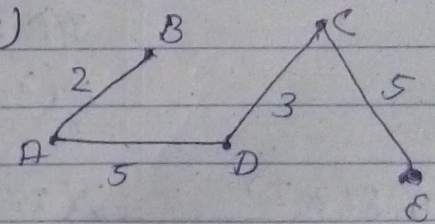
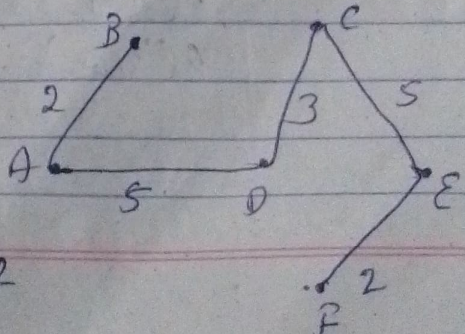
Tree Vertices	Remaining Vertices	Illustration
1) a(-,-)	b(a,5) c(a,7) d(-,∞) e(a,2)	
2) e(a,2)	b(e,4) c(e,5) d(e,3)	
3) b(e,3)	d(b,5) c(e,4)	
4) c(e,4)	d(e,4)	

Cost of MST = 4+4+2+3 = 13 units

3 Apply Prim's algorithm to the following graph and find minimum spanning tree for the given graph

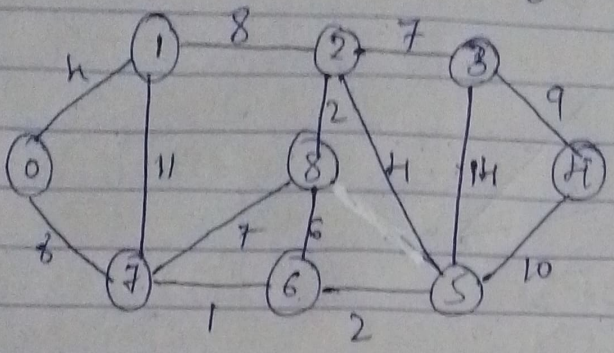


⇒

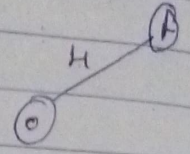
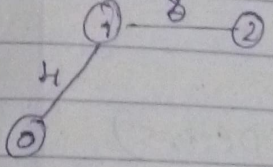
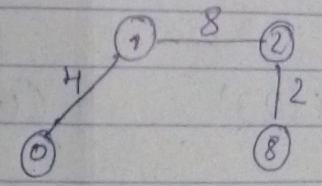
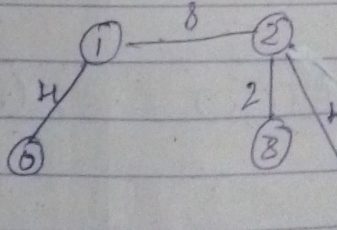
	Tree Vertices	Remaining Vertices	Illustration
1>	A(-,-)	B(A,2) C(-,∞) D(A,5) F(A,6) E(-,∞)	
2>	B(A,2)	C(B,8) D(A,5) E(-,∞) F(A,6)	
3>	D(A,5)	C(D,3) E(D,6) F(A,6)	
4>	C(D,3)	E(C,5) F(A,6)	
5>	E(C,5)	F(E,2)	

cost of MST = 2+5+3+5+2
= 17 units

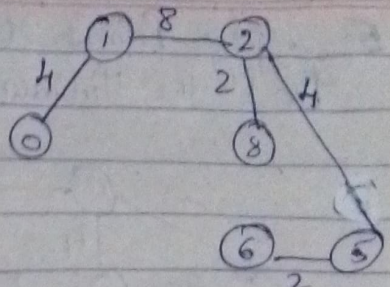
4. Apply Prim's algorithm to the following graph and find the Minimum spanning tree for the given graph



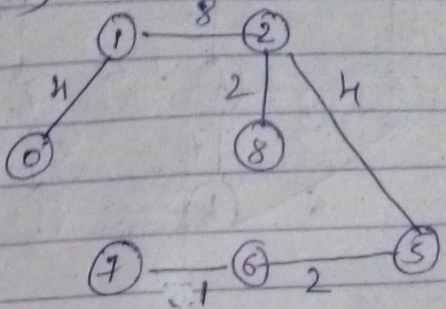
$\Rightarrow$

Tree Vertices	Remaining Vertices	Illustration
1) 0(-,-)	1(0,4) 2(-,∞) 3(-,∞) 4(-,∞) 5(-,∞) 6(-,∞) 7(0,8) 8(-,∞)	
2) 1(0,4)	2(1,8) 3(-,∞) 4(-,∞) 5(-,∞) 6(-,∞) 7(0,8) 8(-,∞)	
3) 2(1,8)	3(2,7) 4(-,∞) 7(0,8) 5(2,14) 8(2,2) 6(-,∞	
4) 8(2,2)	3(2,7) 4(-,∞) 5(2,14) 7(0,8) 6(8,6)	

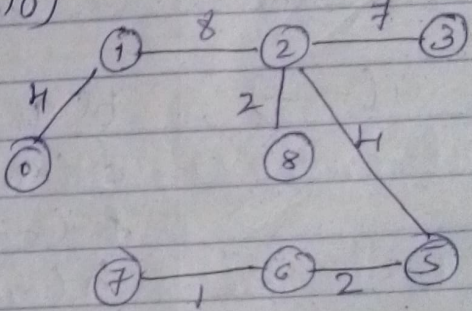
5) 5 (2,4) 3(2,7) 6(5,2)
4(5,10) 7(0,8)



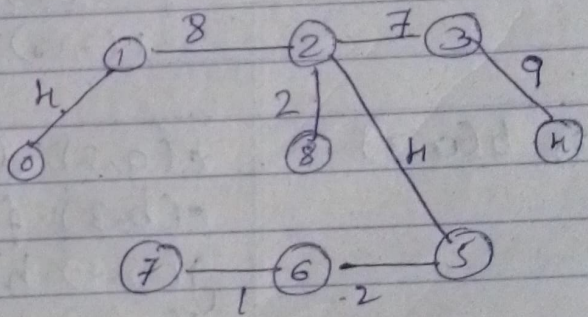
6) 6(5,2) 3(2,7) 4(5,10)
7(6,1)



7) 7(6,1) 3(2,7) 4(5,10)



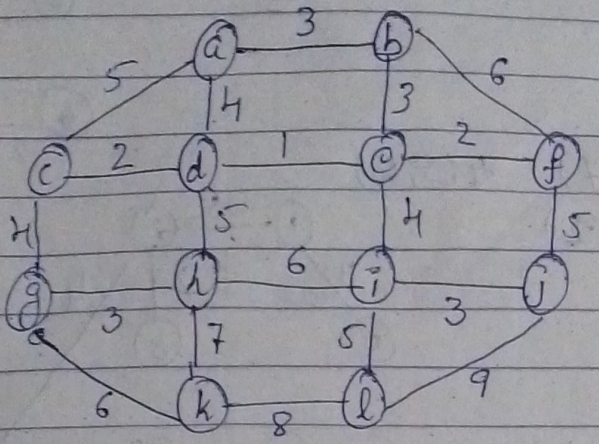
8) 3(2,7) 4(3,9)



Cost of Minimum Spanning Tree

4+8+2+4+1+2+7+9 = 37 units

5. Apply Prim's Algorithm to the following graph and find the Minimum Spanning tree for the given graph.

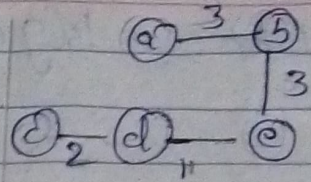


⇒

Tree Vertices	Remaining Vertices	Illustration
1) a(-,-)	b(a,3) c(a,5) d(a,4) e(-,∞) f(-,∞) g(-,∞) h(-,∞) i(-,∞) j(-,∞) k(-,∞) l(-,∞)	a —3— b
2) b(a,3)	c(a,5) d(a,4) e(b,3) f(b,6) g(-,∞) h(-,∞) i(-,∞) j(-,∞) k(-,∞) l(-,∞)	a —3— b, b —3— e
3) e(b,3)	c(a,5) d(e,1) f(e,2) g(-,∞) h(-,∞) i(e,4) j(-,∞) k(-,∞) l(-,∞)	a —3— b, b —3— e, d — e

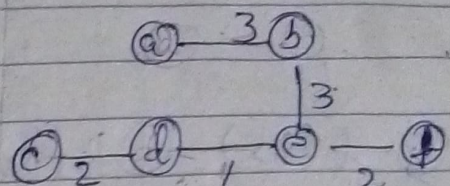
4) d(e,1)

c(d,2) f(e,2)
g(-,∞) h(d,5)
i(e,4) j(-,∞)
k(-,∞) l(-,∞)



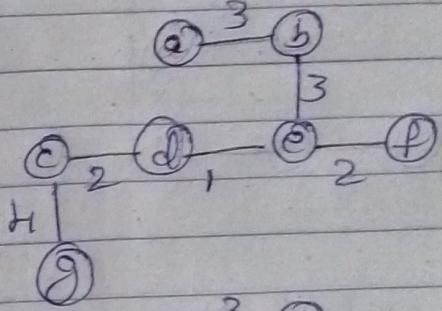
5) e(d,2)

f(e,2) g(e,4)
h(d,5) i(e,4)
j(-,∞) k(-,∞)
l(-,∞)



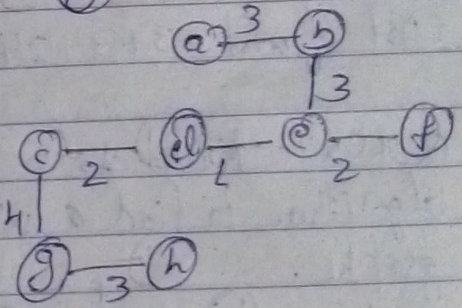
6) f(e,2)

g(e,4) h(d,5)
j(f,5) i(e,4)
l(-,∞) k(-,∞)



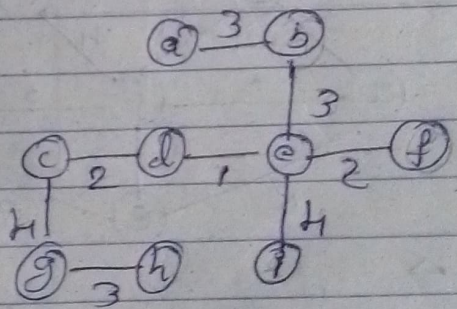
7) g(e,4)

h(g,3) i(e,4)
j(f,5) k(g,6)
l(-,∞)



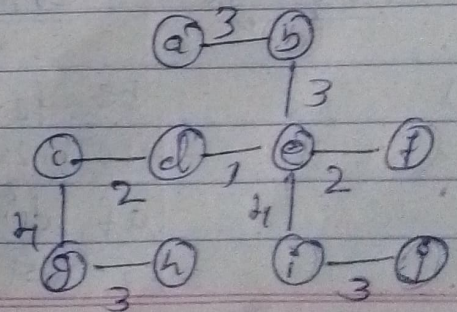
8) h(g,3)

i(e,4) j(f,5)
k(g,6) l(-,∞)

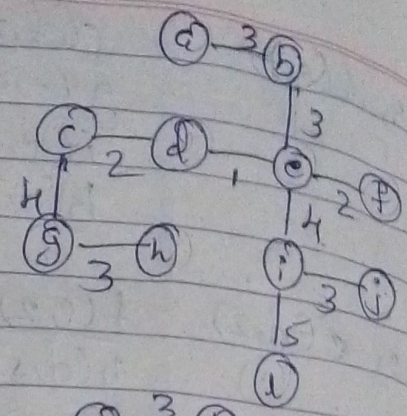


9) i(e,4)

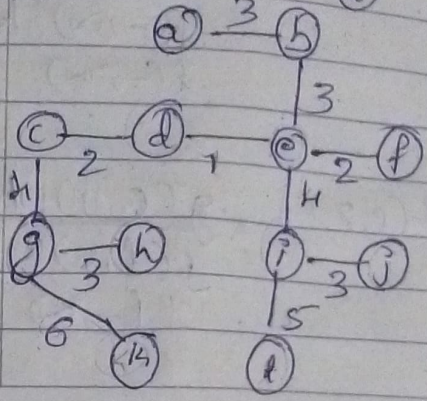
j(i,3) k(g,6)
l(i,5)



10) j(i,3) k(g,6) l(i,5)

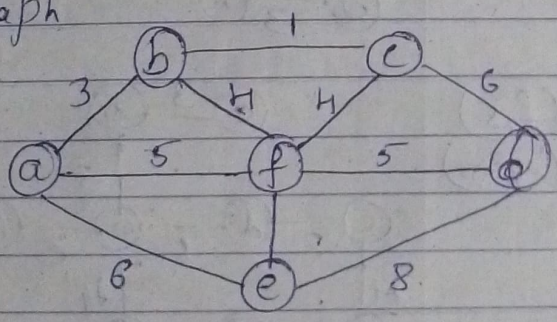


11) l(i,5) k(g,6)



cost of mst = 3+3+2+4+3+1+4+2+5+3+6 = 36 units

6. Write Kruskal's algorithms and apply Kruskal's algorithm to find a minimum spanning tree of the following graph

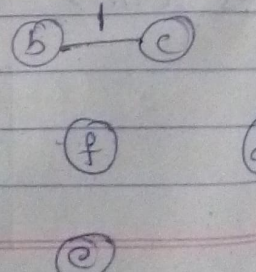


⇒

Tree edge — sorted list of edges — Illustration

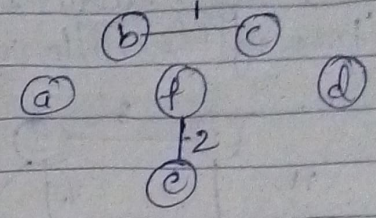
11)

bc	fe	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8



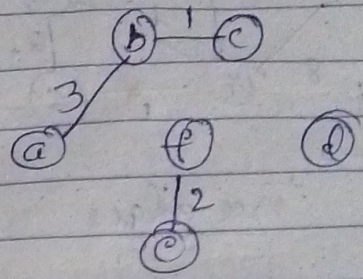
2) bc
1

bc	fe	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8



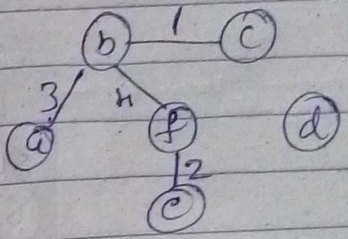
3) fe
2

bc	fe	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8



4) ab
3

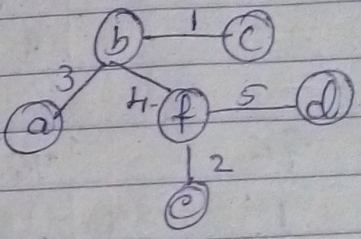
bc	fe	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8



5) bf
4

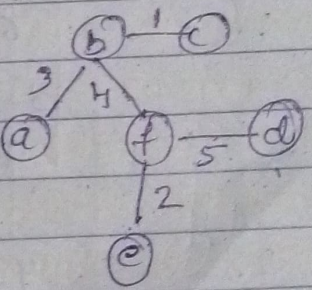
bc	fe	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8

skip



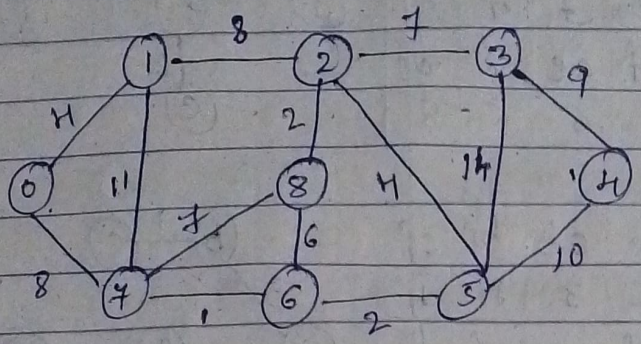
6) df
5

bc	fe	ab	bf	cf
1	2	3	4	4
af	df	ae	cd	de
5	5	6	6	8

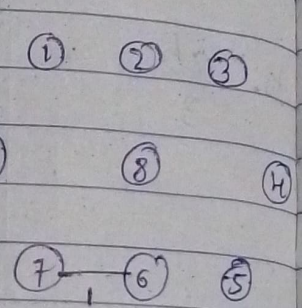
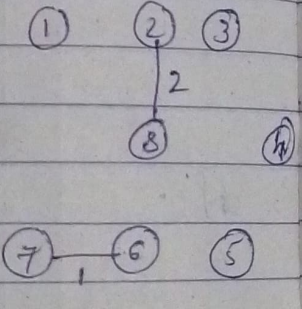
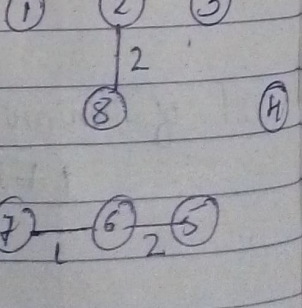


cost of Minimum Spanning Tree
$1+4+3+5+2 = 15$ units

7. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.

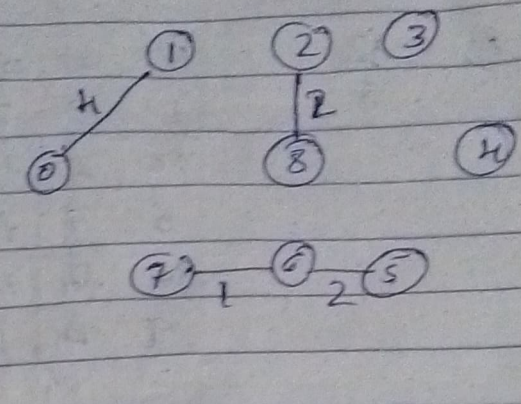


⇒

Tree edges	Sorted list of edge	Illustration
1)	76 28 56 01 25 1 2 2 4 4	
	86 23 78 12 07 6 7 7 8 8	
	34 54 17 35 9 10 11 14	
2) 76 1	76 28 56 01 25 1 2 2 4 4	
	86 23 78 12 07 6 7 7 8 8	
	34 54 17 35 9 10 11 14	
3) 28 2	76 28 56 01 25 1 2 2 4 4	
	86 23 78 12 07 6 7 7 8 8	
	34 54 17 35 9 10 11 14	

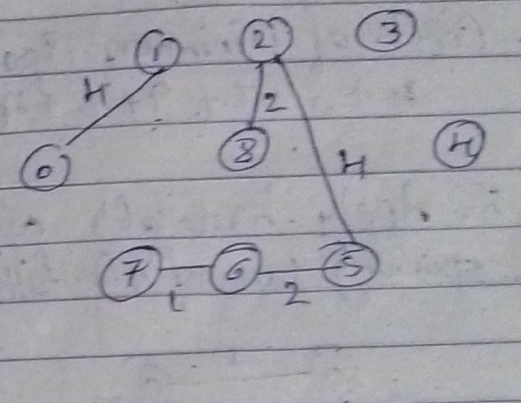
4) 56
2

76	28	56	01	25
1	2	2	4	4
86	23	78	12	07
6	7	7	8	8
34	54	17	35	
9	10	11	14	



5) 01
4

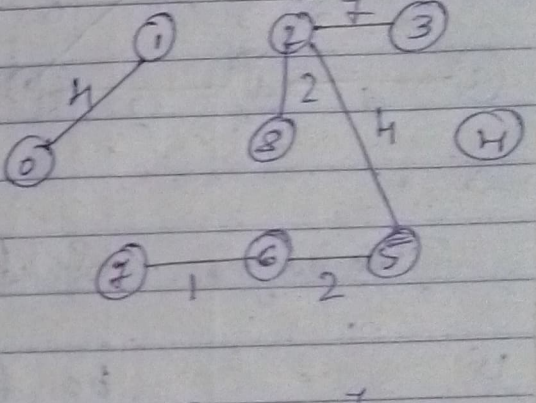
76	28	56	01	25
1	2	2	4	4
86	23	78	12	07
6	7	7	8	8
34	54	17	35	
9	10	11	14	



6) 25
4

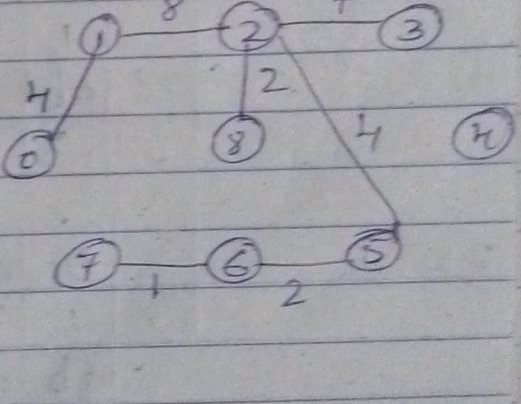
76	28	56	01	25
1	2	2	4	4
86	23	78	12	07
6	7	7	8	8
34	54	17	35	
9	10	11	14	

skip



7) 23
7

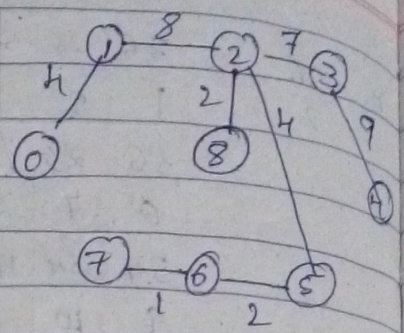
76	28	56	01	25
1	2	2	4	4
86	23	78	12	07
6	7	7	8	8
34	54	17	35	
9	10	11	14	



8) 12
8

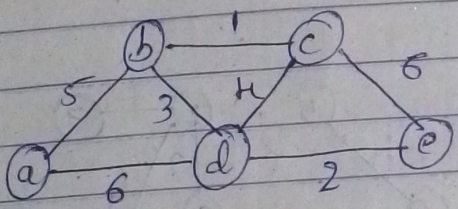
78	28	56	01	25
1	2	2	4	4
86	23	78	12	07
6	7	7	8	8
34	54	17	35	
9	10	11	14	

skip skip skip

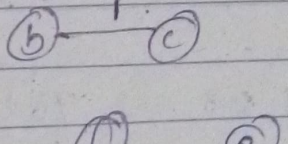
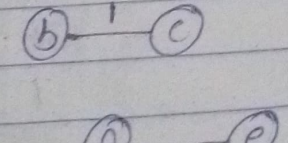


Cost of Minimum Spanning Tree
$4+8+7+9+2+4+2+1 = 37$ units

8. Apply kruskal's algorithm to find a minimum spanning tree of the following graph.

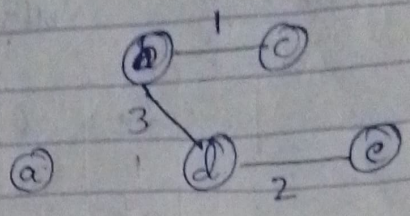


⇒

Tree edges	Sorted list of Edges	Illustration
1)	bc de bd cd 1 2 3 4 ab ad ce 5 6 6	
2)	bc de bd cd 1 2 3 4 ab ad ce 5 6 6	

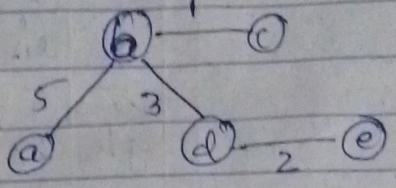
3)

bc	de	bd	cd
1	2	3	4
ab	ad	ce	
5	6	6	



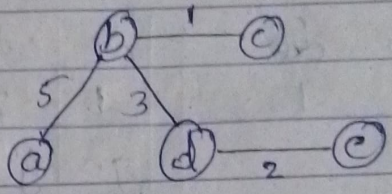
4)

bc	de	bd	cd
1	2	3	4
ab	ad	ce	
5	6	6	



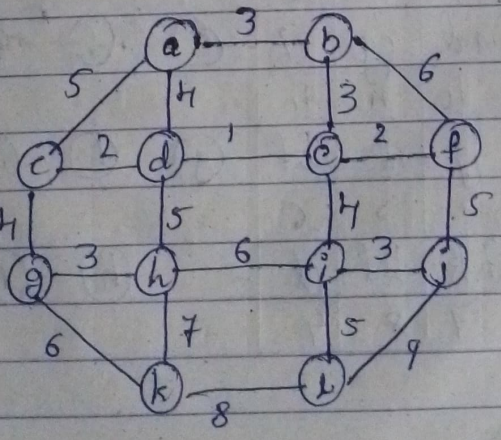
5)

bc	de	bd	cd
1	2	3	4
ab	ad	ce	
5	6	6	



Cost of Minimum Spanning Tree
$5+3+2+1 = 11$ units

9. Apply kruskal's algorithm to find a minimum spanning tree of the following graphs.

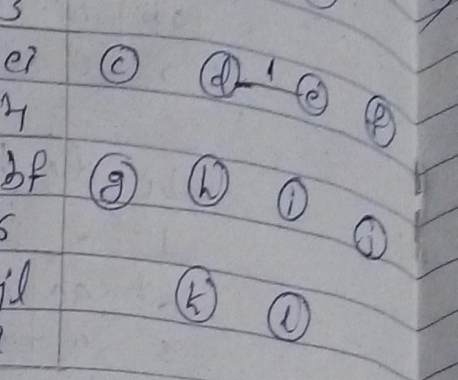


⟹

Tree edge | Sorted list of edge | Illustration

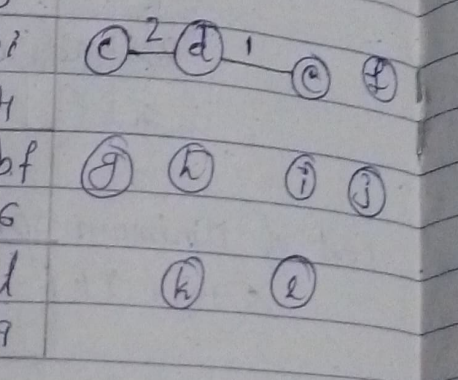
1)

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



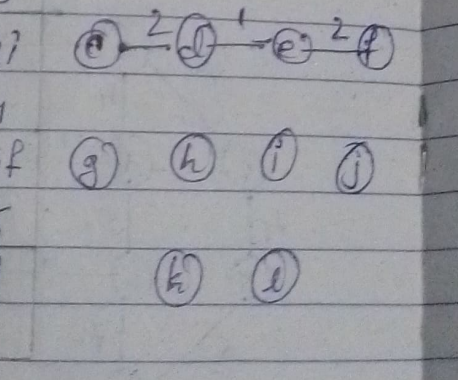
2) de
1

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



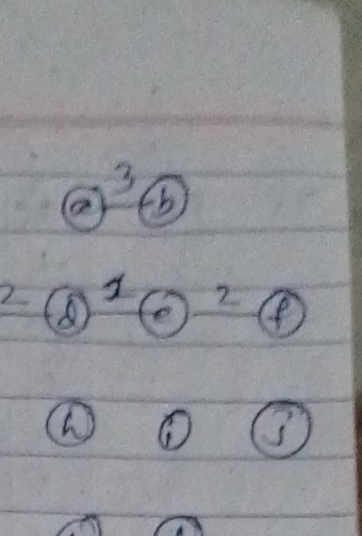
3) cd
2

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



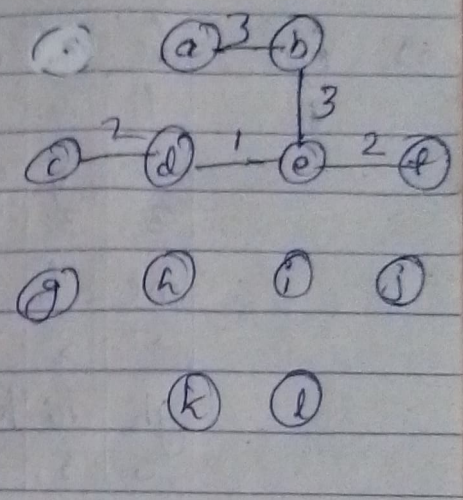
4) ef
2

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



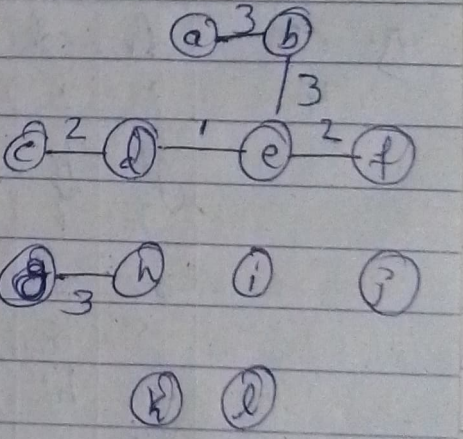
5) ab
3

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



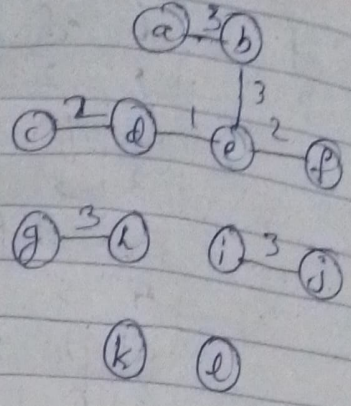
6) be
3

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



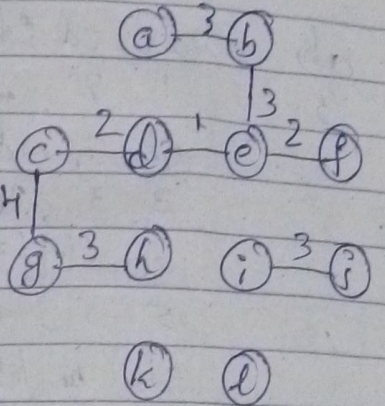
7) gh 3

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



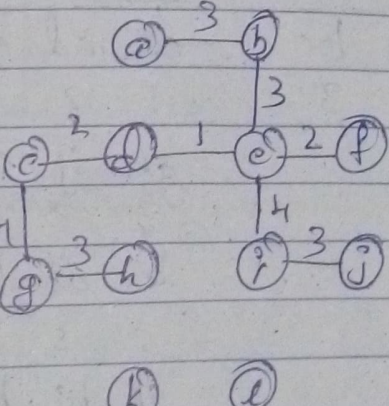
8) ij 3

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



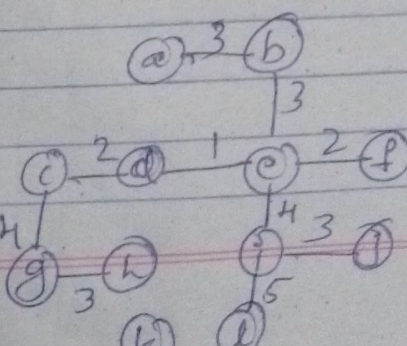
9) cg 4

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



10) ei 4

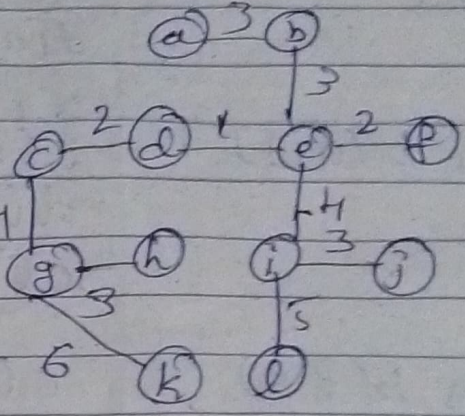
de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6



gk	hi	hk	kl	jl
6	6	7	8	9

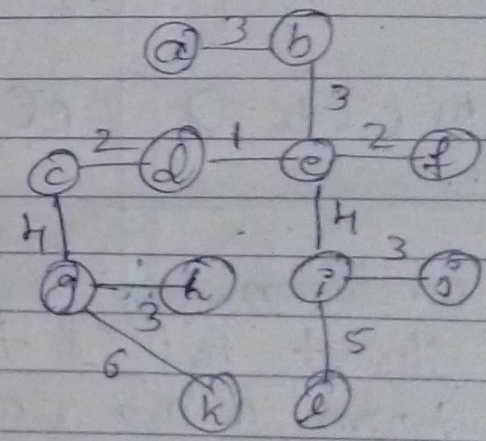
11) il
5

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9



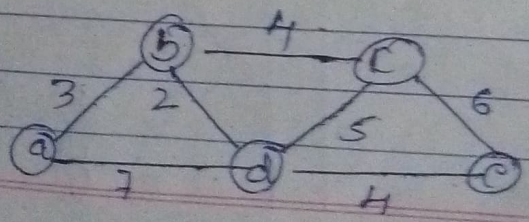
12) gk
6

de	cd	ef	ab	be
1	2	2	3	3
gh	ij	ad	cg	ei
3	3	4	4	4
ac	dh	fj	il	bf
5	5	5	5	6
gk	hi	hk	kl	jl
6	6	7	8	9

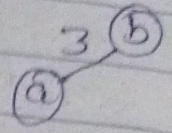
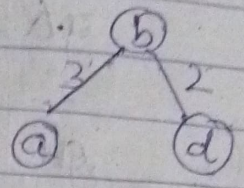
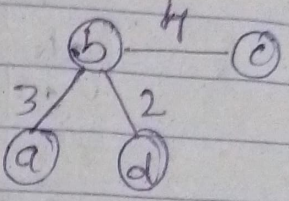
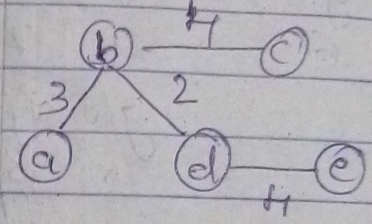


Cost of Minimum spanning tree
$3+3+1+2+2+4+3+6+4+3+5 = \underline{36 \text{ units}}$

10. Write Dijkstra's Algorithm and solve the following instances of the single-source shortest-paths problem with vertex "a" as the source

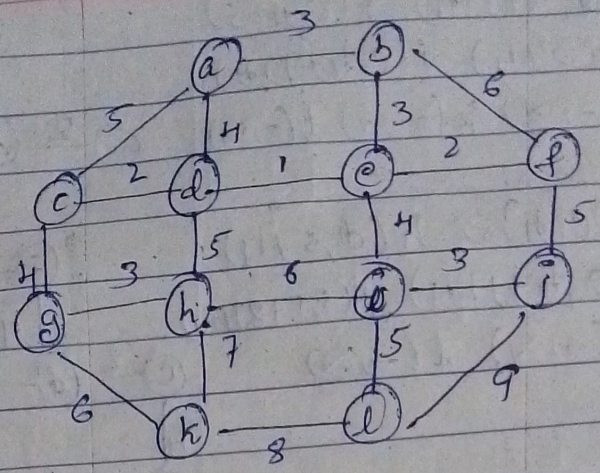


→

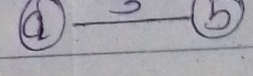
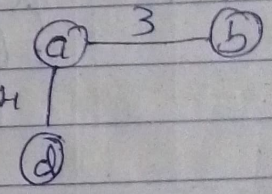
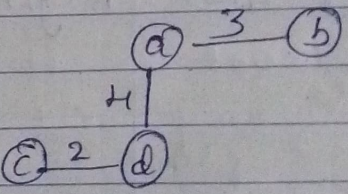
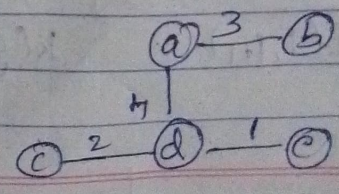
Tree Vertices	Remaining Vertices	Illustration
1) a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)	
2) b(a,3)	c(b, 3+4=7) d(b, 3+2=5) e(-,∞)	
3) b(d,5)	c(b, 7+4=7) e(d, 3+2+4=9)	
4) c(b,7)	e(d, 3+2+4=9)	

Conclusion,
Shortest Path & their length
From a to b : a-b of length 3
From a to c : a-b-c of length 7
From a to d : a-b-d of length 5
From a to e : a-b-d-e of length 9

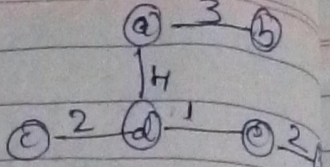
11. Using Dijkstra's Algorithm to solve the following instances of the single-source shortest-paths problem with vertex "a" as the source



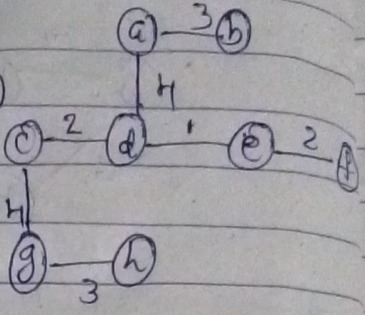
⇒

Tree Vertex	Remaining Vertices	Illustration
1) a(-,0)	b(a,3) c(a,5) d(a,4) e(-,∞) f(-,∞) g(-,∞) h(-,∞) i(-,∞) j(-,∞) k(-,∞) l(-,∞)	
2) b(a,3)	c(a,5) d(a,4) e(b,3+3) f(b,6+3) g(-,∞) h(-,∞) i(-,∞) j(-,∞) k(-,∞) l(-,∞)	
3) d(a,4)	c(a,5) e(d,4+1) f(b,6+3) g(-,∞) h(d,5+4) i(-,∞) j(-,∞) k(-,∞) l(-,∞)	
4) c(a,5)	e(d,5) f(b,6+3) g(c,4+5) h(d,5+4) i(-,∞) j(-,∞) k(-,∞) l(-,∞)	

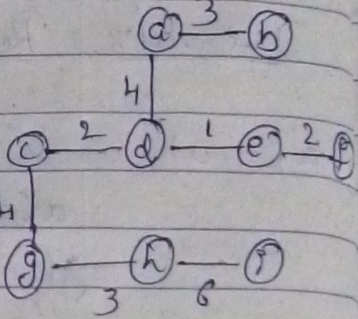
5. c(d,5) f(a, 2+1+4) g(c, 4+5) h(d, 5+4) i(e, 4+1+4) j(-,∞) k(-,∞) l(-,∞)



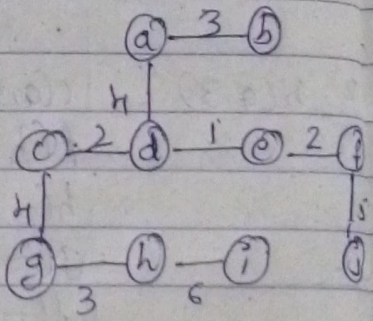
6) f(e,7) g(c, 4+5) h(d, 5+4) i(e, 4+1+4) j(5, 5+2+1+4) k(-,∞) l(-,∞)



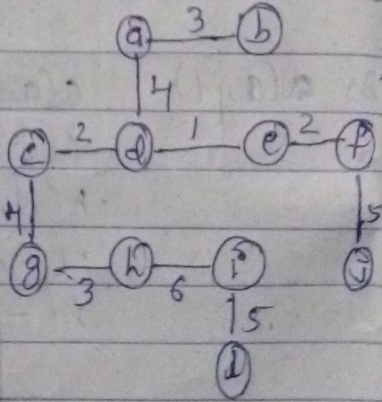
7) h(d,9) i(e,9) j(f,12) k(g,15) l(-,∞)



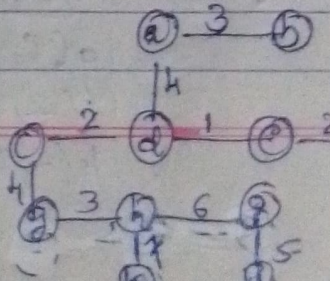
8) i(e,9) j(f,12) k(g,15) i(5, 5+4+1+4)



9) j(f,12) k(g,15) l(i, 5+4+1+4)



10) l(i,14) k(g,15)



Therefore shortes path and their length →

From a to b : a-b of length 3
From a to c : a-d-c of length 6
From a to d : a-d of length 4
From a to e : a-d-e of length 5
From a to f : a-d-e-f of length 7
From a to g : a-d-c-g of length 10
From a to h : a-d-c-g-h of length 13
From a to i : a-d-c-g-h-i of length 19
From a to j : a-d-e-f-j of length 12
From a to k : a-d-c-g-h-k of length 20
From a to l : a-d-c-g-h-i-l of length 14

12. Write an explain the Huffman algorithm and construct Huffman coding tree. Consider the five character alphabet [A, B, C, D] with the following occurance probabilities

characters	A	B	C	D	—
probability	0.35	0.1	0.2	0.2	0.15

⇒

1) 0.1 B + 0.15 — | 0.2 C | 0.2 D | 0.35 A

2) 0.2 C + 0.2 D → 0.25 (0.1 B + 0.15 —) | 0.35 A

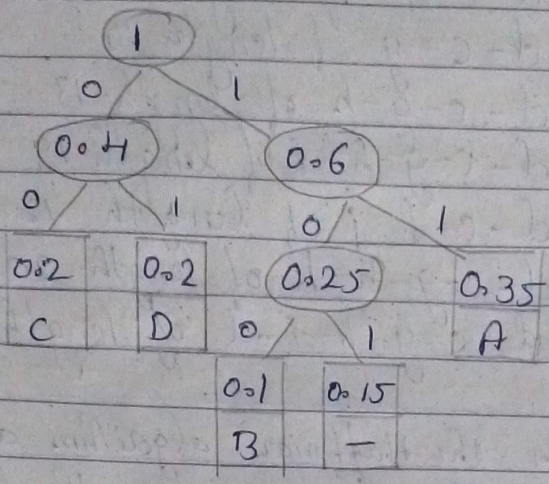
3) 0.25 (0.1 B + 0.15 —) + 0.35 A → 0.4 (0.2 C + 0.2 D)

4) 0.4 + 0.6

0.2	0.2	0.25	0.35
C	D		A

0.1	0.15
B	—

5)



Symbol	A	B	C	D	—
Frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101
Encode	DAD = 011101				

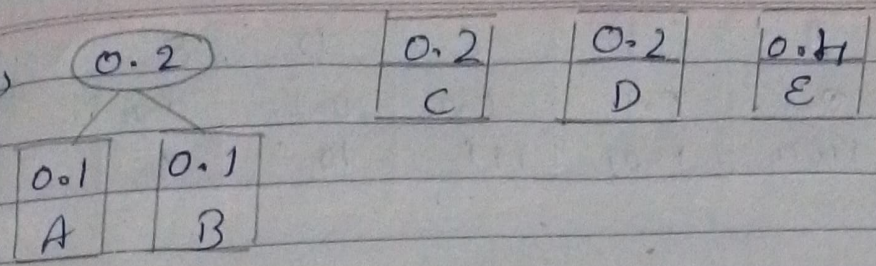
13. Construct a Huffman code for following data

Symbol	A	B	C	D	E
Frequency	0.1	0.1	0.2	0.2	0.4

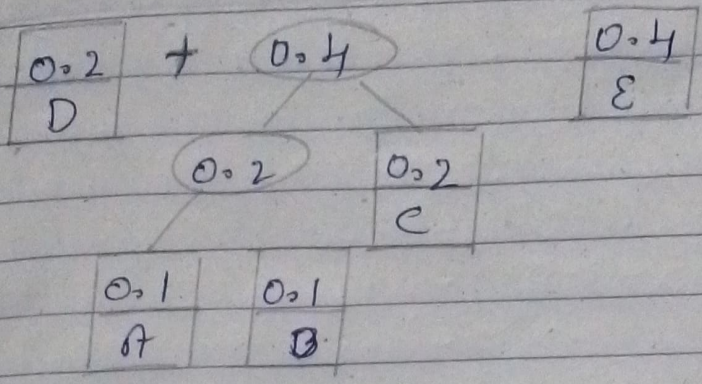
⟹

0.1	+	0.1	0.2	0.2	0.4
A		B	C	D	E

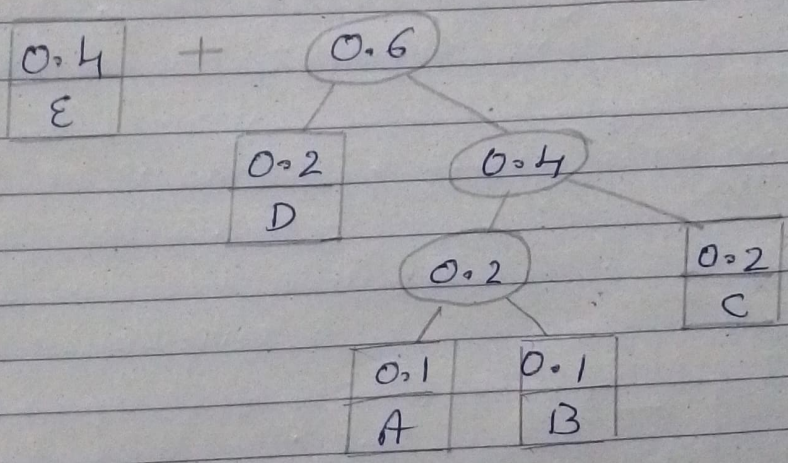
2)


0.2
0.1
A
0.1
B
0.2
C
0.2
D
0.4
E

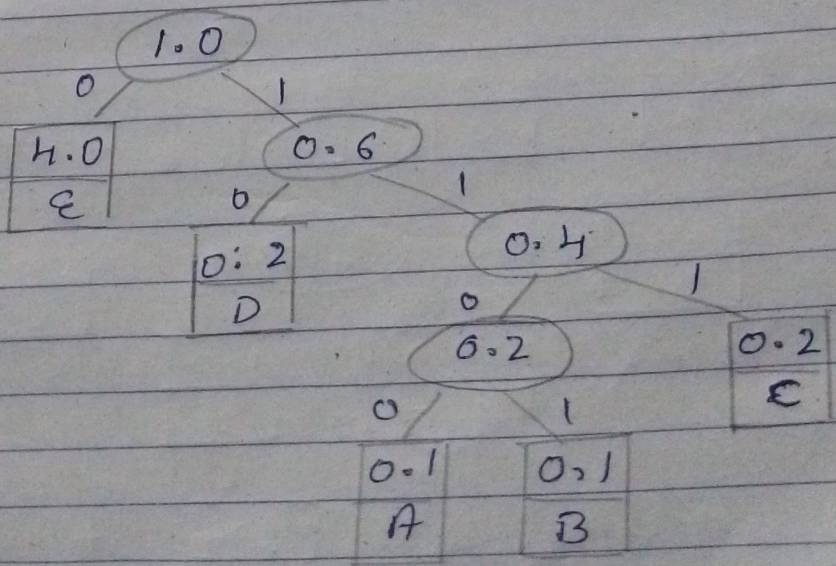
3)


0.2
D
+
0.4
0.2
0.2
C
0.1
A
0.1
B
0.4
E

4)


0.4
E
+
0.6
0.2
D
0.4
0.2
0.2
C
0.1
A
0.1
B

5)


1.0
0
1
4.0
E
0.6
0
1
0.2
D
0.4
0
1
0.2
E
0
1
0.1
A
0.1
B

Symbol	A	B	C	D	E
Frequency	0.1	0.1	0.2	0.2	0.4
Codeword	1100	1101	111	10	0