CBCS SCHEME

USN | | | | | | | | | |

18MCA33

Third Semester MCA Degree Examination, Dec.2019/Jan.2020
Design and Analysis of Algorithms

Time: 3 hrs. Max. Marks: 100

Note: ***Answer FIVE full questions, choosing ONE full question from each module.***

Module-1

1 a. Explain the fundamentals of algorithmic problem solving with a neat diagram. **(10 Marks)**

b. Define the asymptotic notations. **(06 Marks)**

c. Compare the orders of growth of the following using limits:

i) $\log_2 n$ and $\sqrt{n}$ ii) $n!$ and 2^n **(04 Marks)**

OR

2 a. Explain the mathematical analysis of recursive algorithms with an example of Tower of Hanoi puzzle. **(10 Marks)**

b. If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$. Prove that $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ **(06 Marks)**

c. Explain the principal ways of representing graphs for computer algorithms. **(04 Marks)**

Module-2

3 a. Explain bubble sort algorithm with its efficiency. **(06 Marks)**

b. Discuss divide and conquer strategy for designing algorithms. Apply it for multiplication of large integers. **(08 Marks)**

c. Write pseudocode for merge sort. **(06 Marks)**

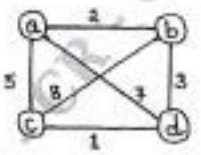
OR

4 a. Explain quick sort algorithm with its efficiency. Trace the algorithm for the following input: 5, 3, 1, 9, 8, 2, 4, 7 **(10 Marks)**

b. Design an algorithm for string matching problem using brute force technique. Apply it to search a pattern ABABC in the text BAABABABCCA. **(06 Marks)**

c. Apply exhaustive search for travelling salesman problem to the given graph in Fig.Q.4(c). **(04 Marks)**

Fig.Q.4(c)



Module-3

5 a. Explain Prim's algorithm with its efficiency. Trace the algorithm for the graph given in Fig.Q.5(a). **(10 Marks)**

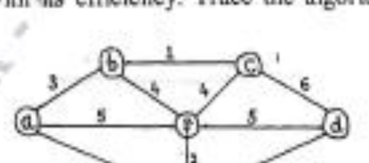


Fig.Q.5(a)

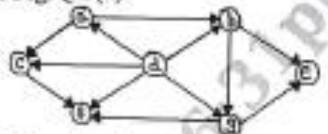
b. Write Johnson-Trotter algorithm. Apply it to generate permutations for n = 3. **(06 Marks)**

c. Explain decrease and conquer algorithm design technique. **(04 Marks)**

OR

6 a. Write an algorithm for DFS traversal. Explain how DFS can be used to solve topological sorting with the graph shown in Fig.Q.6(a). **(10 Marks)**

Fig.Q.6(a)



b. Construct Huffman tree for the following data:

Character	A	B	C	D	-
Probability	0.35	0.1	0.2	0.2	0.15

Encode DAD and Decode 10011011011101 **(06 Marks)**

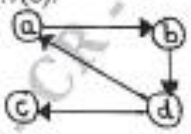
c. Differentiate DFS and BFS. **(04 Marks)**

Module-4

7 a. Write Horspool's string matching algorithm. Apply it to search the pattern BARBER in the given text
JIM_SAW_ME_IN_A_BARBERSHOP **(10 Marks)**

b. Explain the Warshall's algorithm for computing transitive closure. Apply the algorithm for the following digraph shown in Fig.Q.7(b). **(10 Marks)**

Fig.Q.7(b)

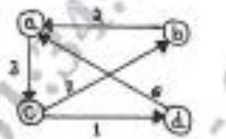


OR

8 a. Explain the comparison counting sort algorithm with its efficiency. Sort the elements 13, 11, 12, 13, 12, 12 by using distribution counting method. **(10 Marks)**

b. Explain Floyd's algorithm with pseudocode and find all-pairs shortest path for the given diagraph Fig.Q.8(b). **(10 Marks)**

Fig.Q.8(b)



Module-5

9 a. Explain P, NP and NP-Complete problems. **(08 Marks)**

b. Draw a decision tree to sort three elements by insertion sort and find its lower bound. **(06 Marks)**

c. Apply backtracking to solve the subset sum problem for the instance
S = {5, 7, 8, 10} and d = 15 **(06 Marks)**

OR

10 a. Explain how backtracking can be used to solve n-queens problem. Find the solution of 4-queens problem using Board's symmetry. **(10 Marks)**

b. Explain branch and bound technique. Solve the following assignment problem:

$$\begin{array}{c|cccc} & J1 & J2 & J3 & J4 \\ \hline A & 9 & 2 & 7 & 8 \\ B & 6 & 4 & 3 & 7 \\ C & 5 & 8 & 1 & 8 \\ D & 7 & 6 & 9 & 4 \end{array}$$

(10 Marks)

* * * * *

CMR INSTITUTE OF TECHNOLOGY

Third Semester MCA Degree Examination Sec 2010/ Jan 2020

Sub:	**Design and Analysis of Algorithms**							**Code:**	**17MCA33**
Date:	*06-02-2020*	*Duration:*	*3hrs*	*Max Marks:*	*100*	*Sem:*	**III Sem**	*Branch:*	*MCA*

Dr. Vakula Rani

Answer any five of the following **100 Marks**

Q1(a) Explain the fundamentals of algorithmic problem Solving with a neat diagram

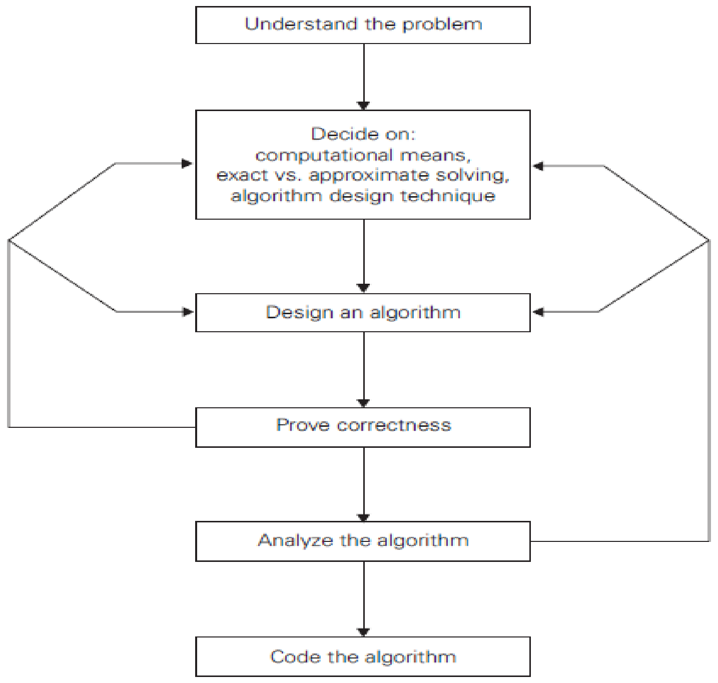


Fig : Algorithm Design and Analysis Process

1. **Understanding the problem:**

Before designing algorithm, one should understand the problem correctly. This may require the problem to be read multiple times, asking questions if required and working out smaller instances of problem by hand. Any input to an algorithm specifies an instance or event of the problem. So, it is very important to set the range of inputs so that the algorithm works for all legitimate inputs i.e work correctly under all circumstances..

2. **Ascertaining the capabilities of a computational Device:**

After understanding the problem, one must think of the machines that execute instructions. The machines that are capable of executing the instructions one after the other is known as sequential machines and algorithms which run on these machines are known as sequential algorithms

Newer machines can run instructions concurrently re known as parallel machines and algorithms which have written for such machines are called parallel algorithms.

If we are dealing with the small problems, we need not worry about the time and memory requirements. But some complex problems which involve processing large amounts of data in real time are required to know about the time and memory requirements where the program is to be executed on the machine.

3. **Choosing between exact and approximate problem solving:**

The algorithms which solves the problem and gives the exact solution is known as Exact Algorithm and one which gives approximate results is known as Approximation Algorithms.

There are two situations in which we may have to go for approximate solution:

i) If the quantity to be computed cannot be calculated exactly. For example finding square roots, solving non linear equations etc.

ii) Complex algorithms may have solutions which take an unreasonably long amount of time if solved exactly. In such a case we may opt for going for a fast but approximate solution.

4. **Deciding on data structures:**
   Algorithms use different data structures for their implementation. Some use simple ones but some other may require complex ones. But, Data structures play a vital role in designing and analyzing the algorithms.
5. **Algorithm Design Techniques:**
   An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. These techniques will provide guidance in designing algorithms for new problems. Various design methods for algorithms exist, some of which are – divide and conquer, dynamic programming, greedy algorithms etc.
6. **Methods of specifying an Algorithm:**
   Algorithm can be specified using natural language and psuedocode. Due to the inherent ambiguity of the natural language, the most prevelant method of specifying an algorithm is using psuedocode.
7. **Proving an Algorithm's correctness:**

Correctness has to be proved for every algorithm. To prove that the algorithm gives the required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others it can be quite complex. Mathematical Induction is normally used for proving algorithm correctness.

8. **Analyzing an algorithm:**

Any Algorithm must be analysed for its efficiency time and space . Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs. Another desirable characteristic is simplicity. A code which is simple reduces the effort in understanding and writing it and thus leads to less chances of error. Another desirable characteristic is generality. An algorithm can be general if it addresses a more general form of the problem for which the algorithm is to be designed and is able to handle all legitimate inputs.

9. **Coding an algorithm:**

Programming the algorithm by using some programming language. Formal verification by proof is done for small programs. Validity of large and complex programs is done through testing and debugging.

Q 1 (b) Define the Asymptotic Notations

Asymptotic notations are the mathematical notations to express time and space complexity. It represents the runnning time of an algorithm.

Different Notations

1. Big oh Notation
2. Omega Notation
3. Theta Notation

1. **Big oh (O) Notation** : A function t(n) is said to be in O[g(n)], t(n) $\in$ O[g(n)] , if t(n) is bounded above by some constant multiple of g(n) for all large n ie.., there exist some positive constant c and some non negative integer no such that $t(n) \leq cg(n)$ for all $n \geq no$.
   Eg. t(n)=100n+5 express in O notation

   $100n+5 <= 100n + n$ for all $n>=5$

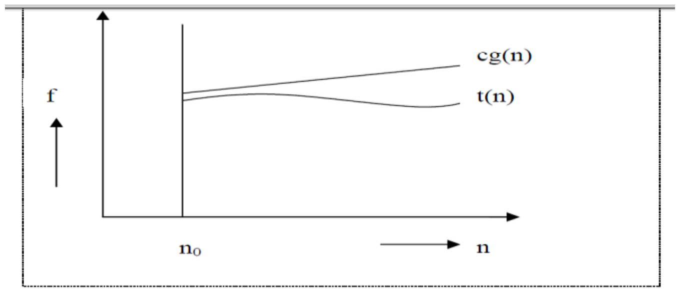
   $<= 101 (n2)$

Let g(n)= n2 ; n0=5 ; c = 101

i.e 100n+5 <=101 n2

t(n) <= c* g(n) for all n>=5

There fore , t(n) ∈ O(n2)



2. **Omega(Ω) -Notation:**

Definition: A function t(n) is said to be in Ω[g(n)], denoted t(n) ∈ Ω[g(n)] , if t(n) is bounded below by some positive constant multiple of g(n) for all large n, ie., there exist some positive constant c and some non negative integer n0 such that

t(n) ≥ cg(n) for all n ≥ n0.

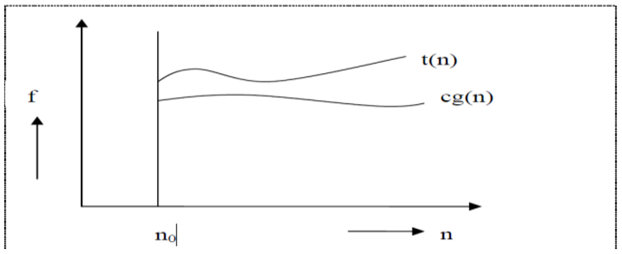
For example:

t(n) = n3 ∈ Ω(n2),

n3 ≥ n2 for all n ≥ n0.

we can select, g(n)= n3 , c=1 and n0=0

t(n) ∈ Ω(n2),



3. **Theta (θ) - Notation:**

Definition: A function t(n) is said to be in θ [g(n)], denoted t(n) ∈ θ (g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n , ie., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that c2g(n) ≤ t(n) ≤ c1g(n) for all n ≥ n0.

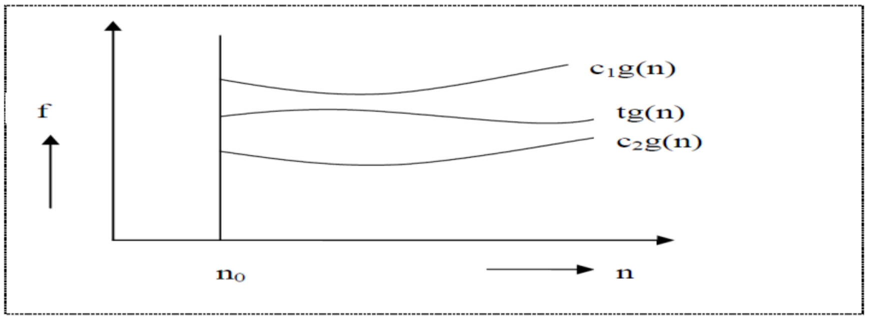
For example 1:

t(n)=100n+5 express in θ notation

100n <= 100n+5 <= 105n for all n>=1

c1=100; c2=105; g(n) = n;

Therefore , t(n) ∈ θ (n)



Q1(c) Compare the order of growth of $(1/2)^{n(n-1)}$ and n^2 using limits.

Sol: The time complexity of a large number of algorithms fall into only a few classes. These classes are listed in Table in increasing order of their orders of growth. Although normally we would expect an algorithm belonging to a lower efficiency class to perform better than an algorithm belonging to higher efficiency classes, theoretically it is possible for this to be reversed. For example if we consider two algorithms with orders (1.001)n and n1000. Then for lot of values of n (1.001)n would perform better but it is rare for an algorithm to have such time complexities.

Class	Name	Comments
n	Linear	Algorithms that scan a list of size n, eg., sequential search, finding the max/min element in an array etc.
$n^{1/2}$	Square root	
logn	Logarithmic	Algorithms in this category are very ef ficient e.g. binary search.
2^n	Exponential	Algorithms that generate all subsets of an n-element set .
n!	factorial	Algorithms that generate all permutations of an n-element set e.g. Travelling Salesman problems

n	logn	$n^{1/2}$	n!	2^n
2	1	1.414	2	4
4	2	2	24	16
8	3	2.824	5760	256

Compare the order of growth of $(1/2)^{n(n-1)}$ and n^2 using limits.

Sol: Using limits for comparing growth of functions we recall that:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0, & if\ f(n) has\ a\ rate\ of\ growth\ less\ than\ g(n) \\ c, & a\ constant\ if\ f(n) and\ g(n) have\ the\ same\ rate\ of\ growth \\ \infty, & if\ f(n) has\ a\ rate\ of\ growth\ greater\ than\ g(n) \end{cases}$$

Finding the limit

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{\frac{1}{2}^{n(n-1)}}{n^2}$$

We notice that

$$\lim_{n\to\infty}\frac{1}{2}^{n(n-1)} = \lim_{n\to\infty} 0.5^{n(n-1)}$$

Since 0.5 < 1, hence if we raise it to higher and higher powers it would keep getting smaller and smaller.
Hence

$\lim_{n\to\infty} 0.5^{n(n-1)}$ = 0

Similarly considering the denominator, $\lim_{n\to\infty} n^2$ will be equal to ∞.

Hence $\lim_{n\to\infty}\frac{\frac{1}{2}^{n(n-1)}}{n^2} = \frac{0}{\infty} = 0$

Q2(a) Explain the mathematical analysis of recursive algorithms with an example of Towers of Hanoi puzzle.

Sol :

A General Plan for Analyzing Efficiency of Recursive Algorithms :

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

2. Identify the algorithm's basic operation.

3. 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.

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

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

For example: consider the recursive algorithm for **Towers of Hanoi problem**

In Towers of Hanoi problem We have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of asmaller one.

To move $n>1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n − 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n − 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg.

```
Algorithm Towers( n,L,M,R)
//Input  : No.of Disks n, three pegs L, M  & R
//Output : the steps to move from L to  R
Begin
    If( n=1)
         Print( " Move disk from L to R")
   Else
       Towers( n-1,L,R,M)
        Print( " Move nth  disk from L to R")
      Towers( n-1,M,L,R)
End
```

Analysis

Let us apply the general plan outlined above to the Tower of Hanoi problem. The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves M(n) depends on n only, and we get the following recurrence equation for it:

$M(n) = M(n − 1) + 1+ M(n − 1)$ for $n > 1$.

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves M(n):

$M(n) = 2M(n − 1) + 1$ for $n > 1$, (2.3)

$M(1) = 1$.

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n − 1) + 1$ sub. $M(n − 1) = 2M(n − 2) + 1$

$= 2[2M(n − 2) + 1]+ 1= 22M(n − 2) + 2 + 1$ sub. $M(n − 2) = 2M(n − 3) + 1$

$= 22[2M(n − 3) + 1]+ 2 + 1= 23M(n − 3) + 22 + 2 + 1$.

The pattern of the first three sums on the left suggests that the next one will be

$24M(n − 4) + 23 + 22 + 2 + 1$, and generally, after i substitutions, we get

$M(n) = 2iM(n − i) + 2i−1 + 2i−2 + . . . + 2 + 1= 2iM(n − i) + 2i − 1$.

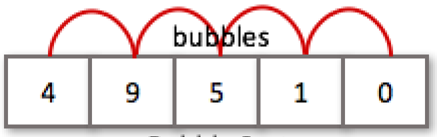
Since the initial condition is specified for $n = 1$, which is achieved for $i = n − 1$, we get the following formula for the solution to recurrence (2.3):

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

Q3(a) Explain bubble sort with its efficiency .

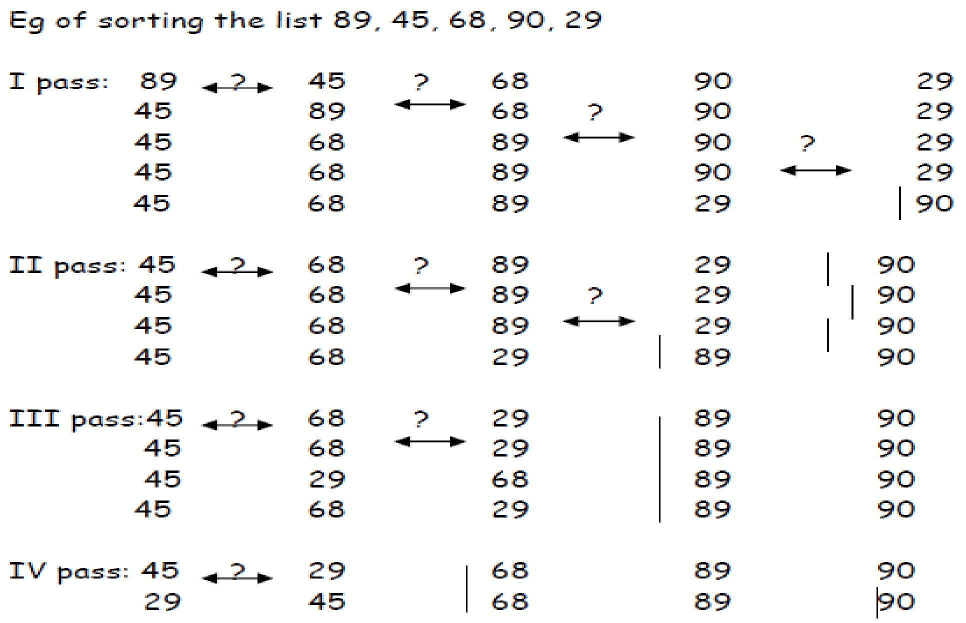
Sol :

Bubble sort compares the value of first element with the immediate next element and swaps according to the requirement and goes till the last element. This iteration repeates for (N - 1) times/steps where N is the number of elements in the list.



Bubble Sort

Compare and swapping two elements like small soap bubbles and hence the name given as bubble sort.



```
Eg of sorting the list 89, 45, 68, 90, 29

I pass:  89 <-?->  45  <-?->  68           90            29
         45        89  <-?->  68  <-?->    90            29
         45        68         89  <-?->    90  <-?->     29
         45        68         89           90  <-?->     29
         45        68         89           29          | 90

II pass: 45 <-?->  68  <-?->  89           29        |  90
         45        68         89  <-?->    29         | 90
         45        68         89  <-?->    29        |  90
         45        68         29        |  89           90

III pass:45 <-?->  68  <-?->  29        |  89           90
         45        68  <-?->  29        |  89           90
         45        29         68        |  89           90
         45        68         29        |  89           90

IV pass: 45 <-?->  29       | 68           89           90
         29        45       | 68           89          |90
```

The algorithm for bubble sort is as follows:

```
ALGORITHM  BubbleSort(A[0..n − 1])
    //Sorts a given array by bubble sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing 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] and A[j + 1]
```

Efficiency :

Bubble sort of N elements can take (N - 1) steps and (N -1) iterations in each steps. Thus resultant is (N - 1)*(N - 1). This sorting algorithm is not however the best in performance when count of the elements are large. Time complexities of bubble sort is O(N^2) [Square of N]. This sorting is well suited for small number of elements and it is easy the implement in C or any other programming languages

The no of key comparisons is the same for all arrays of size n, it is obtained by a sum which is similar to selection sort.

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

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

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

$$= [n(n-1)]/2 \in \Theta(n^2)$$

The no.of key swaps depends on the input. The worst case is same as the no.of key comparisons.

$$C(n) = [n(n-1)]/2 \in \Theta(n^2)$$

Q2(b) If t1(n) $t_1(n) \in O(g_1(n))$ *and* $t_2(n) \in O(g_2(n))$ *then* $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$

if t1(n) $t_1(n) \in O(g_1(n))$ *and* $t_2(n) \in O(g_2(n))$ *then* $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$

PROOF We use the following simple fact about four arbitrary real numbers a1 , b1 , a2, and b2: if a1 < b1 and a2 < b2 then a1 + a2 < 2 max{ b1, b2}.)

This can be proved as follows: adding the two inequalities we get:

a1+a2< b1+b2. - (1)

Without loss of generality, let b1 >= b2. In such a case max(b1,b2) = b1. The inequality (1) becomes

A1+a2 < b1+ b2<= b1+b1 = 2*b1 = 2max(b1,b2).

This proved the above fact.

To prove the main theorem:

Since t1(n) Є O(g1(n)) , there exist some constant c and some nonnegative integer n 1 such that

t1(n) < c1g1 (n) for all n > n1 (According to the definition of O)

Since t2(n) Є O(g2(n)),

t2(n) < c2g2(n) for all n > n2. (According to the definition of O)

Let us denote c3 = max(c1, c2} and consider n > max{ n1 , n2} so that we can use both inequalities. Adding the two inequalities above yields the following:

t1(n) + t2(n) < c1g1 (n) + c2g2(n)

< c3g1(n) + c3g2(n) = c3 [g1(n) + g2(n)]

< c32max{g1 (n),g2(n)}. (According to the fact proved above).

Hence, t1 (n) + t2(n) Є O(max {g1(n),g_2(n)}) (Definition of O)

Graphs Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

A graph is informally thought of a collection of points in a plane called vertices or nodes, some of them connected by line segments called edges or arcs.

Formally, a graph G=<V, E > is defined by a pair of two sets: a finite set V of items called vertices and a set E of pairs of these items called edges.

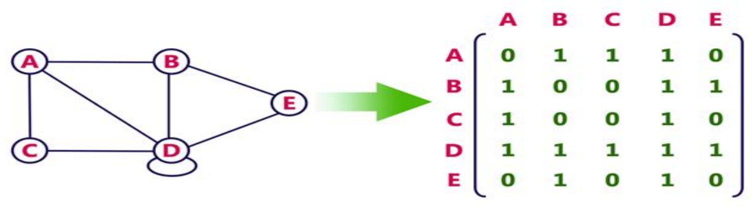
If these pairs of vertices are unordered, i.e. a pair of vertices (u, v) is same as (v, u) then G is undirected; otherwise, the edge (u, v), is directed from vertex u to vertex v, the graph G is directed. Directed graphs are also called digraphs.

There are different ways to represent a graph in memory

1. **Adjacency Matrix**

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i},^{j} = 1$, otherwise $a^{i},^{j}= 0$.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

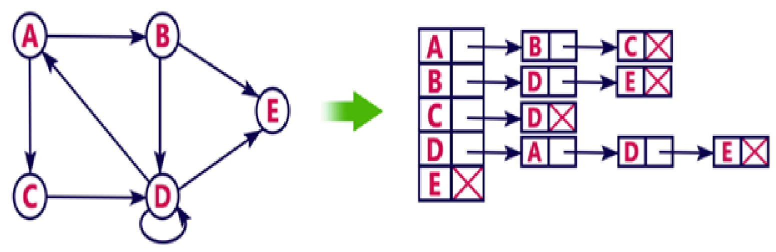
Example: Consider the following undirected graph representation**:**


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

2. **Adjacency List**

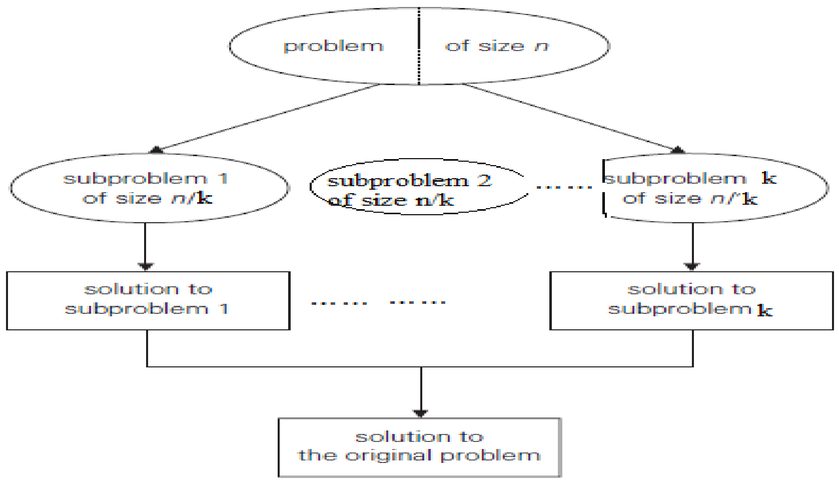
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.

Example : Let's see the following directed graph representation implemented using linked list:



Q3(b) Discuss divide and conquer strategy for designing the algorithms. Apply it for multiply two large integers

Sol:
Divide-and-conquer algorithms work according to the following general plan:
1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.
The general method is shown diagrammatically as below:



Multiplication of Two long Integers:

The conventional algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.)

By using divide-and-conquer method , it would be possible to design an algorithm with fewer than n^2 digit multiplications,

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\ &= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k. \end{aligned}$$

To demonstrate the basic idea of the algorithm, let us start with a case of Four-digit integers – 3421 and 6032 . These numbers can be represented as follows:

X= 3421 = 34 * 10^2 + 21 Let A = 34 ; B = 21

Y=6032 = 60 * 10^2 + 32 Let C = 60; D = 32

Now let us multiply them:

$X * Y = AC * 10^4 + [AC + (A – B) * (D – C) + BD] * 10^2 + BD$

$= 34 * 60 * 10^4 + [(34 * 60) + (34 – 21) * (32 - 60) * 10^2 + (21 * 32)$

$= 2040 * 10000 + [2040 - 364 + 672] * 100 + 672$

$= 20400000 + 234800 + 672$

$= 20635472$

Q3(c) Write Pseudo code for Merge Sort

Sol:

Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A[0..n − 1] by dividing it into two halves A[0.._n/2_ − 1] and A[_n/2_..n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one. The pseudocode for Merge sort is as follows:

```
Algorithm merge(arr,l,mid, u)
    Create a temporary array C[0..u]
    i<-- l
    j <-- mid+1
    k <-- l // index into temporary array
    while i <=mid and j <=u
        if arr[i] <= arr[j]
            C[k] <-- arr[i]
            i <-- i+1
        else
            C[k] <-- arr[j]
            j <-- j+1
    k <-- k+1
//copying rest of elements from first subarray
 while i<=mid
    C[k] <-- arr[i]
    i <-- i+1
    k <-- k+1
//copying rest of elements from second subarray
while j<=u
    C[k] <-- arr[j]
    j <-- j+1
    k <-- k+1
for i in l to u       // copying all elements from temp array to original array
arr[i] <-- C[i]
Algorithm mergesort(arr,l,u)
// only do it if the array contains atleast 2 elements
  if ( l < u )
    mid = (l+u)/2
    mergesort(A,l,mid)
    mergesort(A,mid+1,u)
    Merge(A,l,mid,u)
```

Analysis

We first analyze the merge function used for mergesort. We notice that to merge an array with n elements at every step(in the first three loops) an element is always copied to the temporary array C. Since there are n elements to be copied the number of operations in the first three loops is n. Similarly in the last loop when the elements are copied from temporary array to the original array (arr) there are again “n” copies. Thus the total number of copy operations in the algorithm merge is O(n).

Analyzing the mergesort algorithm we find that each call involves two recursive calls to mergesort with the problem size half and a call to merge which takes O(n) time . Thus the recurrence can be wtitten as:

$$T(n) = 2\ T(n/2)+cn.$$

Applying the master’s method,

a=2, b=2 and d=1.

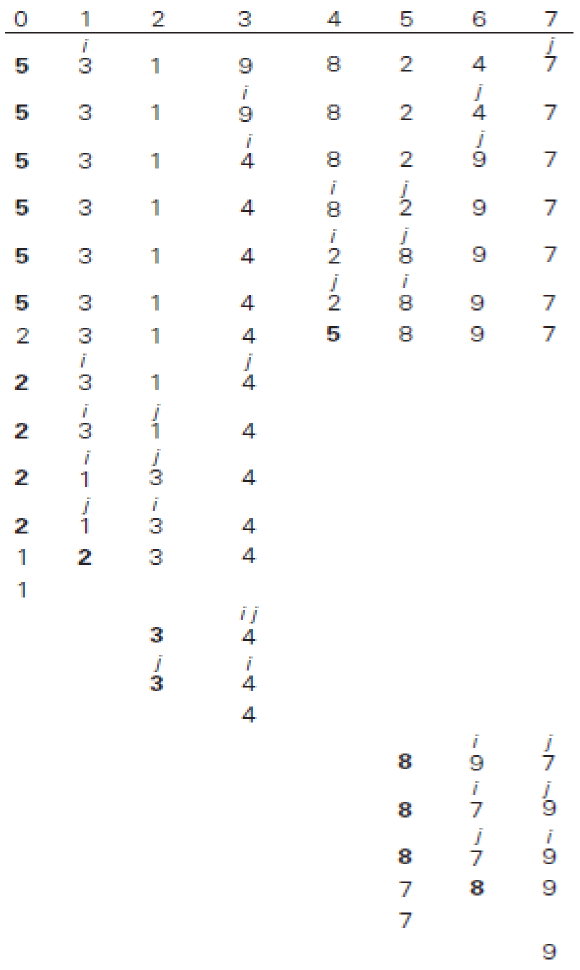
Thus a=bd and thus case 2 of Master’s method applies.

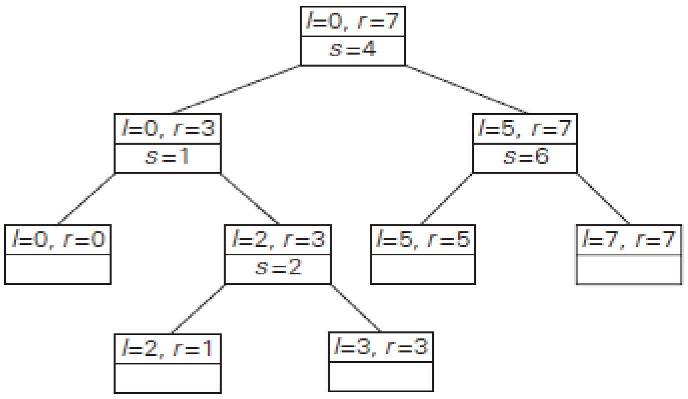
Thus T(n) = O(nlgn).

Q4(a) Explain the Quick sort algorithm with its efficiency. Trace the algorithm for the following Input : 5, 3, 1, 9, 8, 2, 4, 7

QuickSort is a highly efficient sorting algorithm and it uses Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the pivot. Usually, pick first element as pivot. It partitions the large array of data into smaller arrays , one of which holds values smaller than the pivot value and the other holds values greater than the pivot value.

Example : 5, 3, 1, 9, 8, 2, 4, 7





(b)

```
Algorithm Quicksort(A, l,u)
// sort only if there are more than two elements in the array
// Input: Array A[0..n-1] , l lower bound, u Upper bound
//Output : Sorted Array A
Begin
        If ( l < u )
          p<-- partition(A,l,u)
          Quicksort(A,l,p-1)
          Quicksort(A,p+1,u)
End

Algorithm partition(A,l,u)
Begin
        piv <-- A[l]
        i <-- l
        j <-- u +1

        // keep moving i and j till they meet
        repeat
                repeat   i <-- i +1;   until (A[i] >= piv)
                repeat  j <-- j-1 ;   until (A[j]  <= piv)
                if ( i < j)          swap(A[i],A[j])
        until (i>=j)

        swap(A[l],A[j])
  return j
End
```

Analysis:

Analyzing partition we notice that I and j start from the two ends of the array and for each iteration in the algorithm either I moves or j moves. For each move we can have maximum of one swap. Therefore the total number of operations in partition is O(n).

If we consider Quicksort on n elements, then after the partition if one partition has I elements then the other partition has n-i-1 elements(excluding the pivot element). The time taken for quicksort is therefore :

1. The time taken to partition (cn)
2. The time taken for doing quicksort of the first partition
3. The time taken for doing quicksort of the second partition

Thus if T(n) is the time taken by quicksort to sort n elements then
$T(n) = T(i) + T(n-i) + cn$

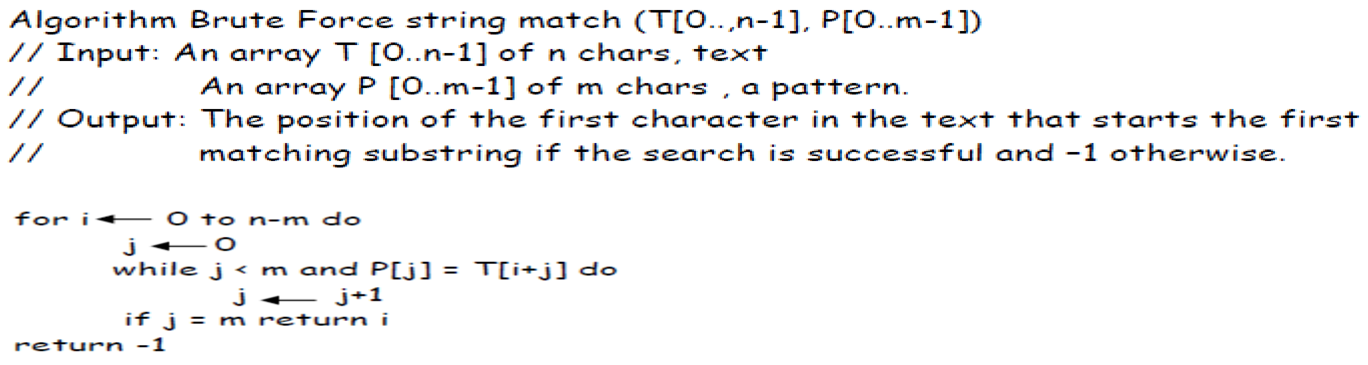
Best Case:

The best case for quicksort occurs when both the partitions are always equal . This happens mostly when the input array is random. In such a case the recurrence becomes
$T(n) = T(n/2)+T(n/2)+cn = 2T(n/2)+cn$
Applying master’s method we find that $T(n) = \theta(n\lg n)$.

Worst case:
In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen for arrays sorted in increasing order. In such a case the recurrence would be
T(n) = T(n-1)+T(0)+n
Assuming T(0) = 0
T(n) = T(n-1)+n.
Using back substitution we find that T(n) = T(n-1)+n = T(n-2)+n-1+n =
T(n-3)+n-2+n-1+n
Expanding till ith step we find
T(n-i) + n-i+1 +….+n
The expansion ends when T(0) is reached. Assigning n-I = 0 => I = n. Substituting in the equation above:
T(n) = T(0) + n-n+1+….n = 1+2…n = n(n+1)/2 = $\theta(n^2)$.
Thus the worst case performance of quicksort is $\theta(n^2)$

Q4(b) Design an algorithm for string matching problem using brute force technique. Apply it to search a pattern ABABC in the BAABABABCCA .

Sol:



```
Algorithm Brute Force string match (T[0..,n-1], P[0..m-1])
// Input: An array T [0..n-1] of n chars, text
//          An array P [0..m-1] of m chars , a pattern.
// Output: The position of the first character in the text that starts the first
//          matching substring if the search is successful and -1 otherwise.

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
```

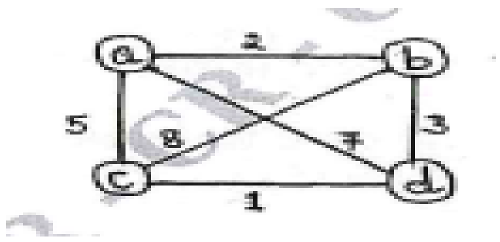
B	**A**	**A**	**B**	**A**	**B**	**A**	**B**	**C**	**C**	**A**
A	**B**	**A**	**B**	**C**						
	A	**B**	**A**	**B**	**C**					
		A	**B**	**A**	**B**	**C**				
			A	**B**	**A**	**B**	**C**			
				A	**B**	**A**	**B**	**C**		

The time complexity would be analyzed by finding the number of times the basic operation j=j+1 is executed. The inner loop will be executed a maximum of m times (j=0 to m-1).
Therefore

T(n)= $\sum_{i=0}^{n-m}\sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-m} m$ = (n-m)*m = $\theta(mn)$.

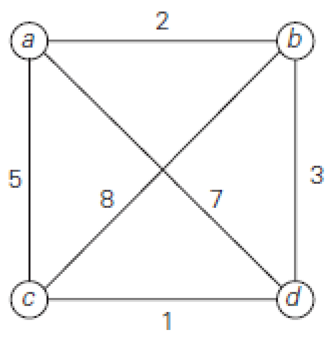
Where m is the length of pattern and n is the length of text.

Q4(c) Apply exhaustive search for Travelling Salesman Problem to the given graph in fig Q4(a).



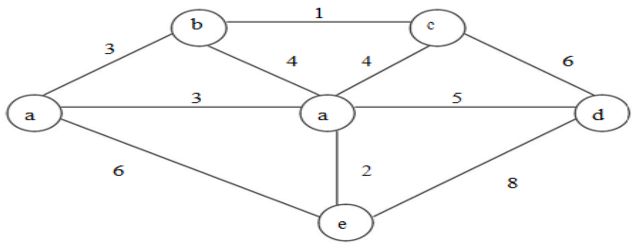
Sol:

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Travelling Salesman Problem (TSP) is viewed as interesting connections to other combinatorial problems. In layman's terms, the problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once



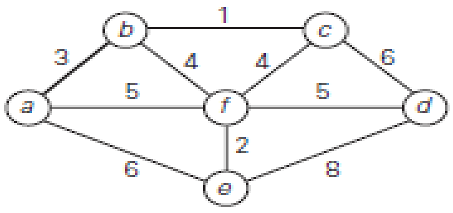
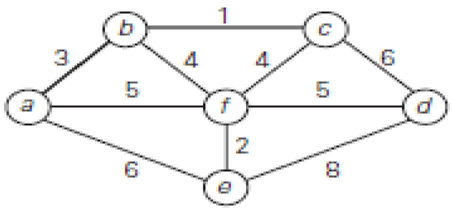
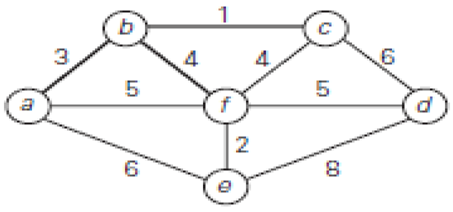
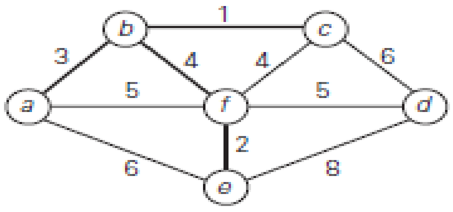
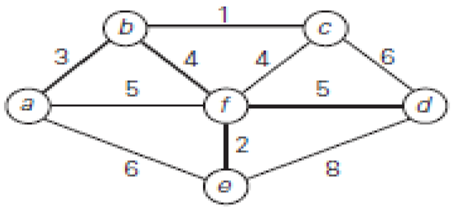
Tour	Length	
a —> b —> c —> d —> a	l = 2 + 8 + 1 + 7 = 18	
a —> b —> d —> c —> a	l = 2 + 3 + 1 + 5 = 11	optimal
a —> c —> b —> d —> a	l = 5 + 8 + 3 + 7 = 23	
a —> c —> d —> b —> a	l = 5 + 1 + 3 + 2 = 11	optimal
a —> d —> b —> c —> a	l = 7 + 3 + 8 + 5 = 23	
a —> d —> c —> b —> a	l = 7 + 1 + 8 + 2 = 18	

Q5(a) Find the minimum cost spanning tree for the given graph below by applying Prim's algorithm. Write the algorithm and compute minimum cost .



Sol:

```
ALGORITHM  Prim(G)
    //Prim's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$   //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Q5(b) . Write Johnson Trotter algorithm . Apply to generate all permutations for n= 3.

Johnson trotter's algorithm works by associating a direction with each element k in a permutation. We indicate such a direction by a small arrow written above the element in question, e.g., $\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{4}\,\overleftarrow{1}$. The element k is said to be mobile in such an arrow-marked permutation if its arrow points to a smaller number adjacent to it. Using the notion of a mobile
element, we can give the following description of the Johnson-Trotter algorithm
for generating permutations.

```
ALGORITHM  JohnsonTrotter(n)
    //Implements Johnson-Trotter algorithm for generating permutations
    //Input: A positive integer n
    //Output: A list of all permutations of {1, . . . , n}
    initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \dots \overleftarrow{n}$
    while the last permutation has a mobile element do
        find its largest mobile element k
        swap k with the adjacent element k's arrow points to
        reverse the direction of all the elements that are larger than k
        add the new permutation to the list
```

We start with permutations. we assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n; more generally, they can be interpreted as indices of elements in an n-element set {a1, . . . , an }.What would the decrease-by-one technique suggest for the problem of generating all n! permutations . The smaller-by-one problem is to generate all (n − 1)! permutations. Assuming that the smaller problem is solved,
we can get a solution to the larger one by inserting n in each of the n possible positions among elements of every permutation of n − 1 elements. All the permutations obtained in this fashion will be distinct (why?), and their total number will be n(n − 1)!= n!. Hence, we will obtain all the permutations of {1, . . . , n}.
We can insert n in the previously generated permutations either left to right or right to left. It turns out that it is beneficial to start with inserting n into 12 . . . (n − 1) by moving right to left and then switch direction every time a new permutation of {1, . . . , n − 1} needs to be processed.
An example of applying this approach bottom up for n = 3
(For the method being discussed, these two elements are always adjacent to each other.

Start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

The advantage of this order of generating permutations stems from the fact that it satisfies the minimal-change requirement: each permutation can be obtained from its immediate predecessor by exchanging just two elements in it.
This algorithm is one of the most efficient for generating permutations; it can be implemented to run in time proportional to the number of permutations, i.e.,
in O(n!).

Q5(c) Explain Decrease-and-Conquer algorithm Design Technique

The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited to give a recursive or iteratively implementation also called ***incremental approach***.

In other words, decrease and conquer can be considered a special case of divide and conquer where the number of subproblems generated is one. There are three major variations of decrease-and-conquer:

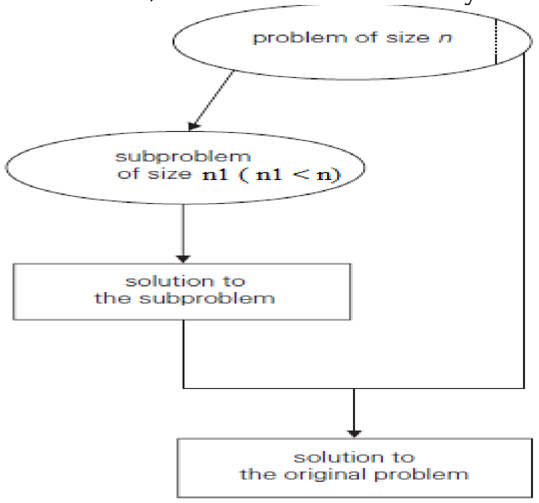
Decrease by a constant: In the *decrease-by-a-constant* variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Example for finding factorial of a number f(n) = f(n-1)*n. Here the problem size is reduced by a constant(here 1) everytime.

Decrease by a constant factor: The *decrease-by-a-constant-factor* technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. For example in binary search the problem size is reduced everytime by a factor of 2.

Variable size decrease: in the *variable-size-decrease* variety of decrease-and- conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. This algorithm is based on the formula

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.


problem of size n
subproblem of size n1 (n1 < n)
solution to the subproblem
solution to the original problem

The general algorithm for decrease and conquer is

```
Algorithm DecreaseAndConquer(P,S) // P is a problem of size n
    Generate a problem $P_1$ of size $n_1 (n_1<n)$
    Solve the Problem $P_1$ and let the solution be $S_1$.
    Use $S_1$ to generate the solution S for the problem P.
```

The time complexity T(n) to solve the problem P would be described by the recurrence: T(n) = T(n1)+f(n) where f(n) is the time required to create solution for P from the solution to problem P1.

For example: consider the problem of generating permutation of numbers 1,2,… n. This problem can be solved using decrease and conquer in the following manner.

Step 1: Generate all permutations of numbers 1,2,… n-1 using the same approach

Step 2: For each permutation generated in the previous step create new permutations by inserting the number n at different positions.(There are n ways of doing this.

The recurrence for the above method would be T(n) = T(n-1) +N(n-1)* n.

Where N(n-1) would be the number of permutations of the numbers 1,2,… n-1

Q6(b) Construct Huffman Tree for the following data . Encode DAD and Decode 100110111110

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

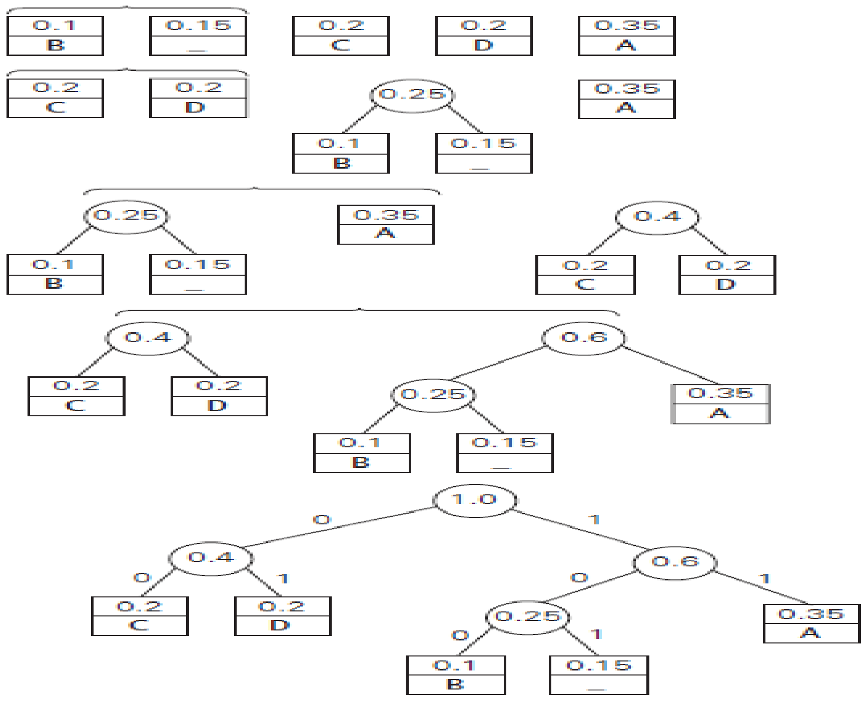
Huffman's algorithm

Step 1 Initialize *n* one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a ***Huffman tree***. It

defines—in the manner described above—a ***Huffman code***.



Encode String - **DAD : 011101**

D	**A**	**D**
01	**11**	**01**

Decode String – 100 11 01 101 1101

100	**11**	**01**	**101**	**11**	**01**
B	**A**	**D**	**-**	**A**	**D**

Q6(c) Differentiate DFS & BFS

Sol:

Sr. No.	**Key**	**BFS**	**DFS**
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.

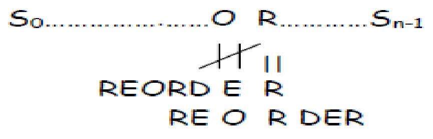
4	Suitablity for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

Q7(a) Write Horspool's string matching algorithm. Apply it to search pattern BARBER in the given text.

Sol:

Horspool's algorithm is used for string matching and performs better than the brute force string matching by attempting the largest possible shift after every mismatch. this however, is done at the cost of extra storage which is a shift table maintained. While matching a string with the pattern the following four cases occur

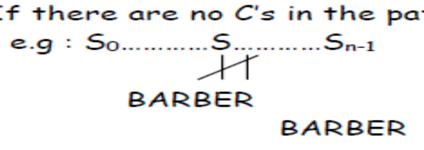
Case 4: If C happens to be the last character in the pattern and there are other C's among its first n-1 characters the shift in same as case 2.



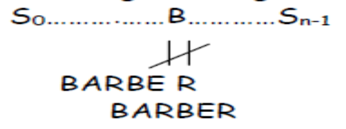
Input enhancement makes repetitive comparisons unnecessary. Shift sizes are precomputed and stored in a table. The shift value is calculated by the formula:

$$t(c)=\begin{cases}\text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \\ \text{the distance from the rightmost } c \text{ among the 1st } m-1 \text{ characters of} \\ \text{the pattern to its last character, otherwise}\end{cases}$$

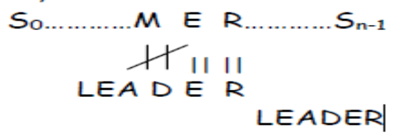
Case 1: If there are no C's in the pattern, shift the pattern by its entire length to right.



Case 2: If there are occurrences of character 'c' in the pattern but the last one, then shift should align the right most occurrence of c in the pattern

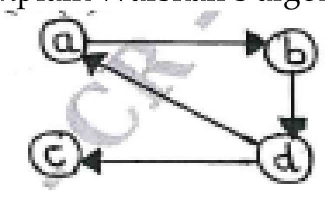


Case 3: If C happens to be the last character in the pattern then there are no C's among its m-1 character, shift same as case 1.



```
Algorithm Shifttable(p[0..m-1])

// Fills the table by Horspool's & Boya-Moore
// Input: pattern p[0..m-1] and an alphabet of possible characters
// Output: Table[0..size-1] indexed by the alphabet's characters and filled with shift
            sizes computed using t(c)
initialize all the elements of Table with m.
for j←0 to n-1 do Table[p[j]] ← m-1-j.
      return table.
```

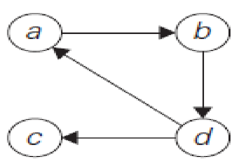
Q7(b) . Explain Warshall's algorithm for the finding the transitive closure of a graph.



Sol:

Adjacency Matrix for the given graph

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in

```
ALGORITHM  Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

Q8(a) Explain the comparison counting algorithm with its efficiency . Sort the elements 13,11,12,13,12,12 by using distribution counting method.

EXAMPLE Consider sorting the array

13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to *n* − 1, the distribution values must be reduced by 1 to get corresponding element positions.
It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position 4 − 1= 3 of the array *S* that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array.

	D[0..2]			*S*[0..5]					
A [5] = 12	1	**4**	6				12		
A [4] = 12	1	**3**	6			12			
A [3] = 13	1	2	**6**						13
A [2] = 12	1	**2**	5		12				
A [1] = 11	**1**	1	5	11					
A [0] = 13	0	1	**5**					13	

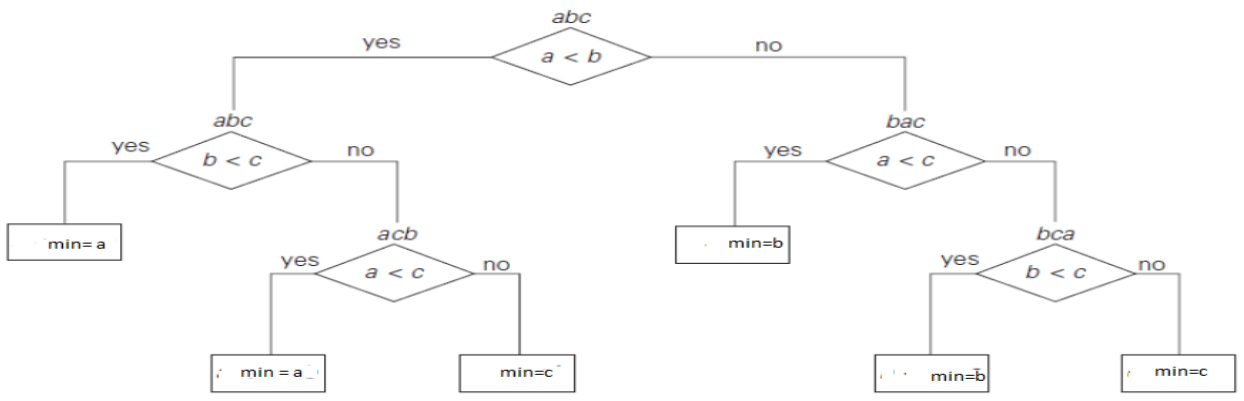
```
Algorithm DistributionCounting (A[0..n-1])

// Sorts an array
//  Input: A[0..n-1] of integers between l & u (l≤u)
// Output: S[0..n-1] of A's elements sorted in increasing order

for j←0 to u-l do D[j]←0                    // initialize frequencies
for i←0 to n-1 do D[A[i]-l]←D[A[i]-l]+1    // compute frequencies
for j←1 to u-l do D[j]←D[j-1]+D[j]         //  reuse for distribution
for i←n-1 down to 0 do
        j  ←  A[i]-l
        S[D[j]-1]  ← A[i]
        D[j]  ← D[j]-1
return s
```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array *A*. This is a better time-efficiency class than that of the most efficient sorting algorithms – mergesort, quicksort, and heapsort.

9.(a) What is Decision Tree ? Obtain the decision tree to find minimum of 3 numbers.

Decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., k < k'.The node's left subtree contains the information about subsequent comparisons made if , k < k', and its right subtree does the same for the case of k > k'. (For the sake of simplicity, we assume throughout this section that all input items are distinct.) Each leaf represents a possible outcome of the algorithm's run on some input of size n. Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons.

The algorithm's work on a particular input of size n can be traced by a path from the root to a leaf in its decision tree, and the number of comparisons made by the algorithm on such a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree. The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, hasto be tall enough to have that many leaves. We use the lemma that for any binary tree with l leaves and height h, h ≥ log2 l. Indeed, a binary tree of height h with the largest number of leaves has all its leaves on the last level. Hence, the largest number of leaves in such a tree is 2h. In other words, 2h ≥ l, which immediately implies, h ≥ log2 l.



Q9(b) Apply back tracking to solve the Sum of subsets problem for the given instance S = { 5, 7, 8, 10} and M = 15

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Clearly $x_1, \ldots, x_k$ cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i's are initially in nondecreasing order. In this case $x_1, \ldots, x_k$ cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

The bounding functions we use are therefore

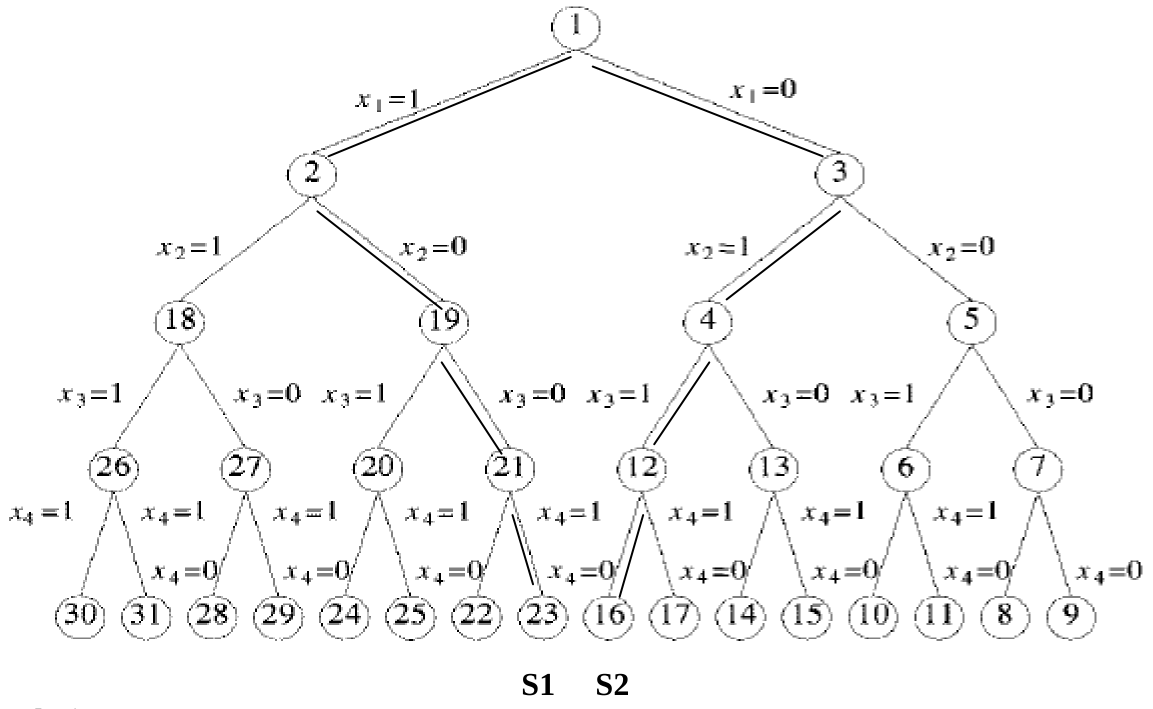
This simplification results from the realization that if $x_k = 1$, then

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

$$B_k(x_1,\ldots,x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \ge m$$

$$\text{and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \le m$$

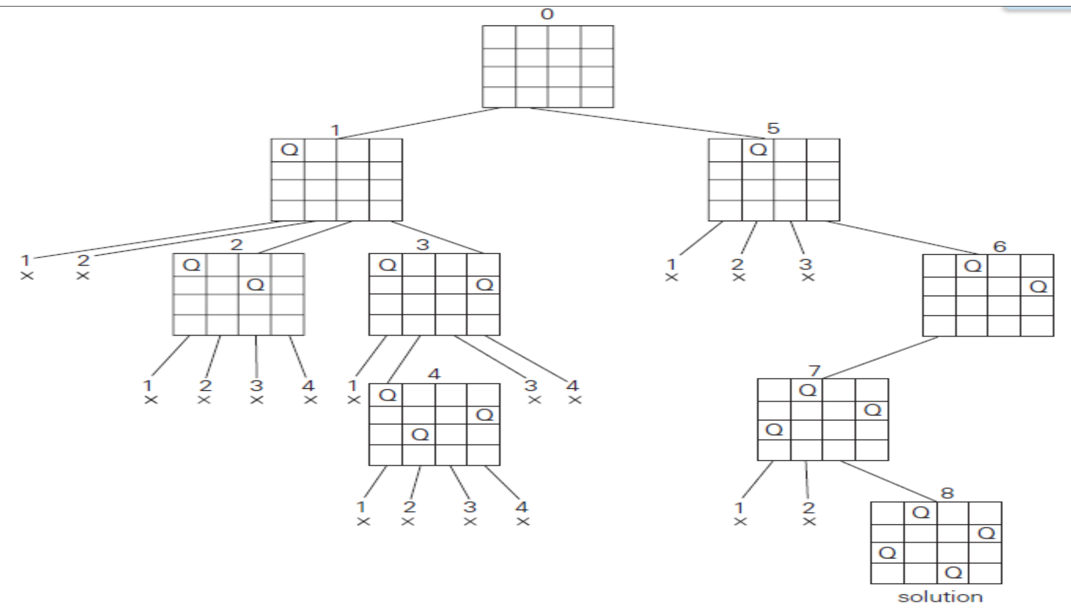
possible organization for the sum of subsets problems.



Solutions:
S1 = (1,0,0,1) = { 5,10}
S2 = (0,1,1,0) = { 7,8}

Q10(a) Explain how Back Tracking can be used to solve N-Queens problem. Find the solution of 4-Queens using border Symmetry.

N Queen's problem:
The n-queens problem. is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
To solve this using backtracing we use the following strategy. Consider the 4 Queens problem:
We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem.
The state space tree is shown below:



Q10 (b) Explain Brach and Bound technique. Solve the following assignment problem which is

$$\begin{array}{c|cccc} & J1 & J2 & J3 & J4 \\ \hline A & 9 & 2 & 7 & 8 \\ B & 6 & 4 & 3 & 7 \\ C & 5 & 8 & 1 & 8 \\ D & 7 & 6 & 9 & 4 \end{array}$$

Sol:

Problem Statement : There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity C[i, j] for each pair i, j = 1, 2, . . . , n. The problem is to find an assignment with the minimum total cost.

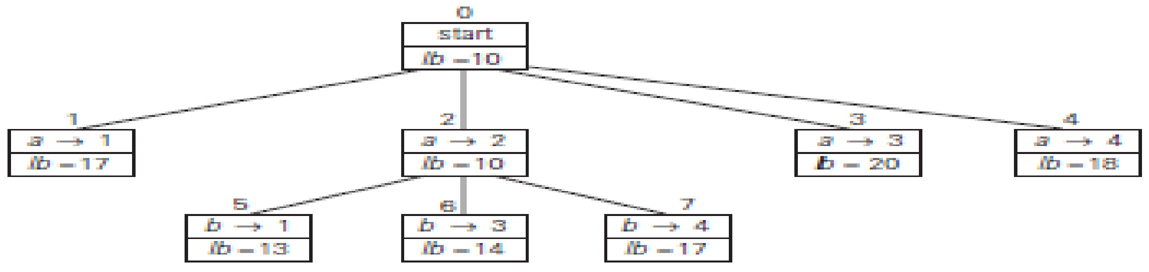
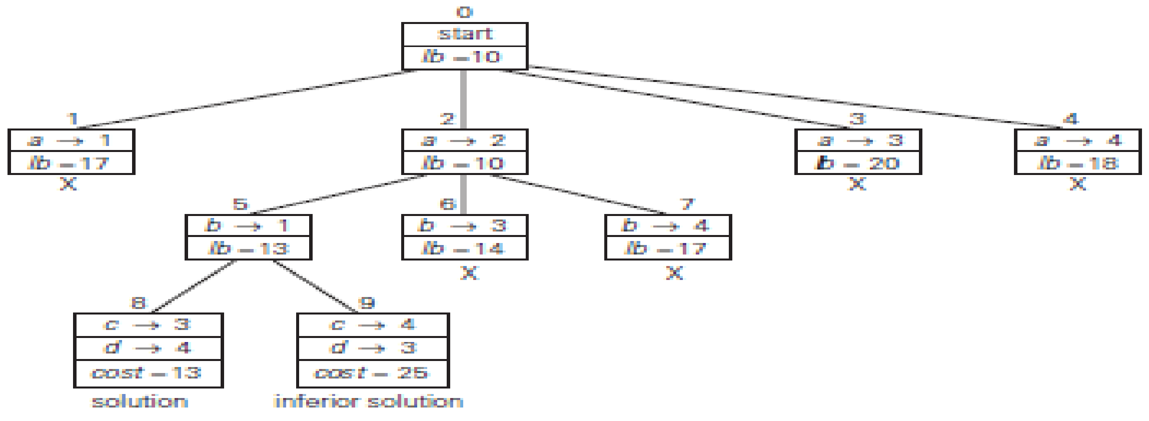


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.



Best solution : a->2, b->1 c->3,d->4 Min Cost = 13