Model Question Paper with effect from 2023-24 (CBCS Scheme)

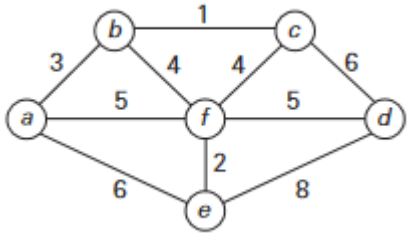
USN | | | | | | | | | | |

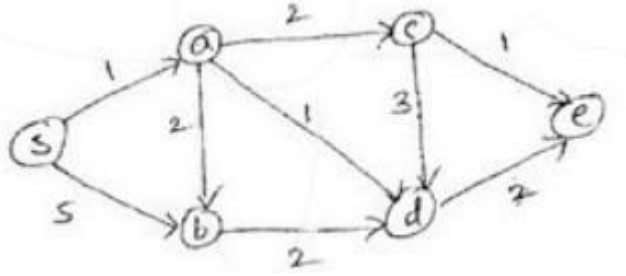
Fourth Semester B.E. Degree Examination

Analysis and Designs of Algorithms

TIME: 03 Hours **Max. Marks: 100**

Note: 01. Answer any **FIVE** full questions, choosing at least **ONE** question from each **MODULE**.

		Module -1	BL		Marks
Q.01	a	Define algorithm. Explain asymptotic notations Big Oh, Big Omega and Big Theta notations	L2		08
	b	Explain the general plan for analyzing the efficiency of a recursive algorithm. Suggest a recursive algorithm to find factorial of number. Derive its efficiency	L2		08
	c	If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then show that $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$.	L2		04
		OR			
Q.02	a	With neat diagram explain different steps in designing and analyzing an algorithm	L2		08
	b	Explain the general plan for analyzing the efficiency of a non-recursive algorithm. Suggest a non-recursive algorithm to find maximum element in the list of n numbers. Derive its efficiency	L2		08
	c	With the algorithm derive the worst case efficiency for Bubble sort	L2		04
		Module-2			
Q. 03	a	Explain the concept of divide and conquer. Design an algorithm for merge sort and derive its time complexity	L2		10
	b	Design an insertion sort algorithm and obtain its time complexity. Apply insertion sort on these elements. 25,75,40,10,20,	L3		10
		OR			
Q.04	a	Explain Strassen's matrix multiplication and derive its time complexity	L2		10
	b	Design an algorithm for quick sort algorithm. Apply quick sort on these elements. 25,75,40,10,20,05,15	L3		10
		Module-3			
Q. 05	a	Define AVL Trees. Explain its four rotation types	L2		10
	b	Construct bottom up heap for the list 2,9,7,6,5,8. Obtain its time complexity	L3		10
		OR			
Q. 06	a	Define heap. Explain the properties of heap along with its representation.	L2		10
	b	Design Horspools algorithm for string matching. Apply Horspools algorithm to find the pattern BARBER in the text: JIM_SAW_ME_IN_A_BARBERSHOP	L3		10
		Module-4			
Q. 07	a	Construct minimum cost spanning tree using Kruskals algorithm for the following graph. 	L3		10
	b	What are Huffman Trees? Construct the Huffman tree for the following data. <table><tr><td>Character</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>-</td></tr><tr><td>Probability</td><td>0.5</td><td>0.35</td><td>0.5</td><td>0.1</td><td>0.4</td><td>0.2</td></tr></table> Encode DAD-CBE using Huffman Encoding.	L3		10

<table>
<tr><td colspan="3">OR</td><td></td><td></td><td></td></tr>
<tr><td>Q. 08</td><td>a</td><td>Apply Dijkstra's algorithm to find single source shortest path for the given graph by considering S as the source vertex.
 
</td><td>L3</td><td></td><td>10</td></tr>
<tr><td></td><td>b</td><td>Define transitive closure of a graph. Apply Warshalls algorithm to compute transitive closure of a directed graph
<table>
<tr><td></td><td>a</td><td>b</td><td>c</td><td>d</td></tr>
<tr><td>a</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>b</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>c</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>d</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
</table></td><td>L3</td><td></td><td>10</td></tr>
<tr><td colspan="3">Module-5</td><td></td><td></td><td></td></tr>
<tr><td>Q. 09</td><td>a</td><td>Explain the following with examples
i) P problem
ii) NP Problem
iii) NP- Complete problem
iv) NP – Hard Problems</td><td>L2</td><td></td><td>10</td></tr>
<tr><td></td><td>b</td><td>What is backtracking? Apply backtracking to solve the below instance of sum of subset problem S={5,10,12,13,15,18} d=30</td><td>L3</td><td></td><td>10</td></tr>
<tr><td colspan="3">OR</td><td></td><td></td><td></td></tr>
<tr><td>Q. 10</td><td>a</td><td>Illustrate N queen's problem using backtracking to solve 4-Queens problem</td><td>L2</td><td></td><td>10</td></tr>
<tr><td></td><td>b</td><td>Using Branch and Bound technique solve the below instance of knapsack problem.
<table>
<tr><td>Item</td><td>Weight</td><td>Value</td></tr>
<tr><td>1</td><td>2</td><td>12</td></tr>
<tr><td>2</td><td>1</td><td>10</td></tr>
<tr><td>3</td><td>3</td><td>20</td></tr>
<tr><td>4</td><td>2</td><td>5</td></tr>
</table>
Capacity 5</td><td>L3</td><td></td><td>10</td></tr>
</table>

Date :

ANALYSIS AND DESIGN OF ALGORITHMS

Model Question Paper - 2023-24

MODULE-1

1. a) Define Algorithm. Explain asymptotic notations Big Oh, Big Omega and Big Theta notations.

Ans:- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e, for obtaining a required output for any legitimate input in a finite amount of time.

Asymptotic Notations:-

To Compare and rank such order of growth, computer scientists use three nototons: O(big oh), Ω(big omega) & Θ(big theta).

O-notation

Definition: A function t(n) is said to be in O(g(n)), denoted $t(n) \in O(g(n))$, if t(n) is bounded above bysome constant multiple of g(n) for all large n, i.e, if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0$$

M T W T F S S

Date :

"THE ONLY WAY TO ACHIEVE THE IMPOSSIBLE IS TO BELIEVE IT IS POSSIBLE"

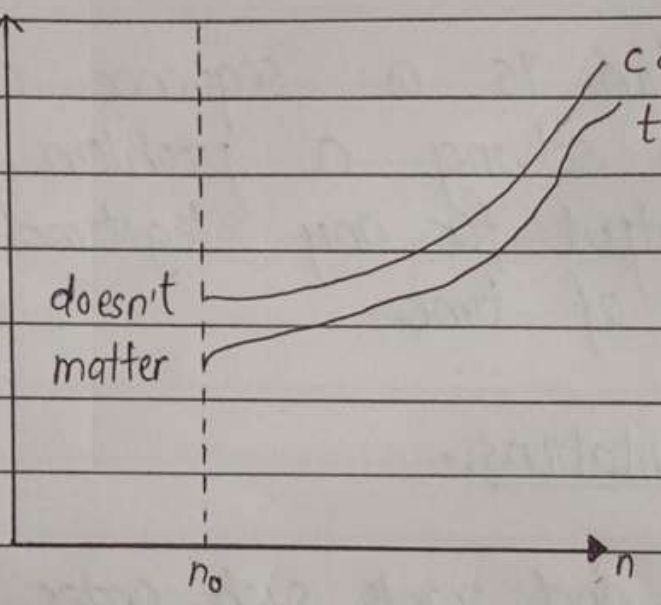


Figure: Big-oh notation : $t(n) \in O(g(n))$

As an example, let us formally prove one of the assertions made in the introduction $100n + 5 \in O(n^2)$. Indeed

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5) = 101n \leq 101n^2$$

Thus, as values of constant c and n_0 required by the definition, we can take 101 and 5, respectively.

M T W T F S S

COMPASS

Date :

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also reason that

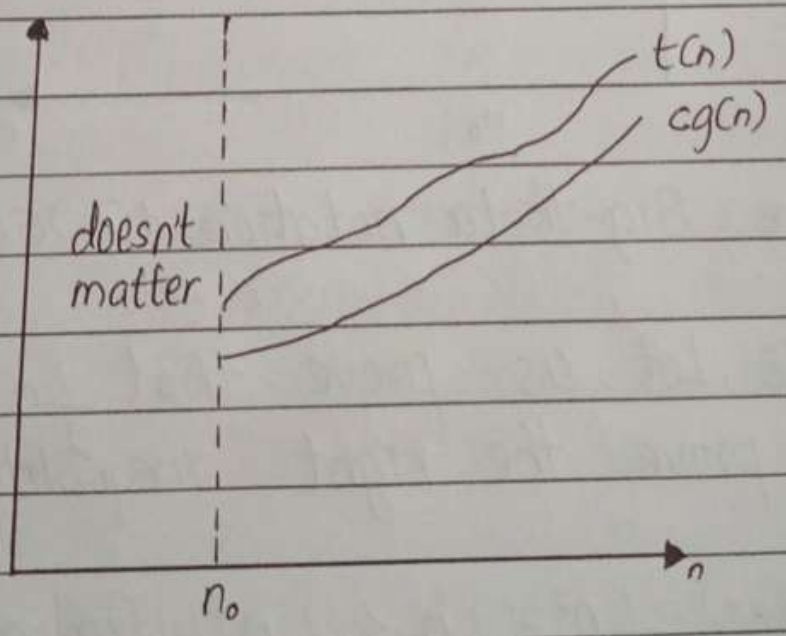
$100n + 5 \leq 100n + 5n$ (for all $n \geq 1$) $= 105n$

to complete the proof with $C = 105$ and $n_0 = 1$.

Ω-notation

Definition: A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, 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 n_0 such that

$t(n) \geq cg(n)$ for all $n \geq n_0$



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

Example of the formal proof that $n^3 \in \Omega(n^2)$:

$n^3 \geq n^2$ for all $n \geq 0$.

i.e, we can select $C = 1$ and $n_0 = 0$.

M T W T F S S

Date :

Θ-notation

definition:- A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, 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 constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$

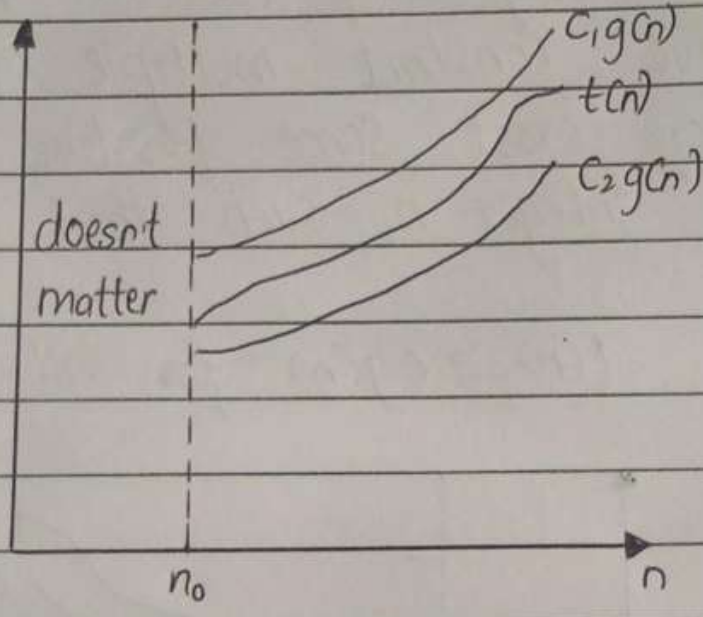


Figure : Big-theta notation: $t(n) \in \Theta(g(n))$.

For example, Let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$.
first, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \text{ for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2)$$

$$= \frac{1}{4}n^2$$

Date :

M T W T F S S

Hence, we can select $C_2 = \frac{1}{4}$, $C_1 = \frac{1}{2}$ and $n_0 = 2$.

b. Explain the general plan for analyzing the efficiency of a recursive algorithm. Suggest a recursive algorithm to find factorial of number. Derive its efficiency.

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating input size
2. Identify algorithm basic operation
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate worst, average and best case efficiency seperately.
4. set up recurrence relation, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. Solve the recurrence.

Factorial Function

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of n!:

$F(n) = F(n-1) * n$ for $n \geq 1$ and

$F(0) = 1$

HARDSHIPS OFTEN PREPARE ORDINARY PEOPLE FOR AN EXTRAORDINARY DESTINY

```
Algorithm Factorial(n)
//Purpose - Compute n! factorial recursively
//Input - A non negative integer n
//Output - The value of n!
{
if (n=0) // base Case
    return 1;
else
    return Factorial (n-1)*n;
}
```

Analysis

1. Input size : given number = n
2. Basic operation : multiplication
3. No best, worst, average cases.
4. Let M(n) denotes number of multiplications.

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0$$
$$M(0) = 0$$

Where : M(n-1) : to Compute Factorial (n-1)
1 : to multiply factorial (n-1) by n

M T W T F S S

COMPASS

Date :

5. Solve the recurrence relation using backwa
Substitution method:

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

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

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

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

$= M(n-3) + 3$

The general formula for the above pattern fo
Some "i" is as follows:

$= M(n-i) + i$

By taking the advantage of the initial condit
given i.e, $M(0) = 0$, we now equate $n-i = 0$ and obta
$i = n$

Substitute $i = n$ in the pattern formula to get the
ultimate result of the backword substitution

$= M(n-n) + n$

$= M(0) + n$

$M(n) = n$

$M(n) \in O(n)$

The number of multiplications to compute the
factorial of n is n where the time complexit
is linear.

6. If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then
Show that $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\}$

M T W T F S S

* Given :-

$t_1(n) \in O(g_1(n))$

$t_2(n) \in O(g_2(n))$

$t_1(n) \leq c_1(g_1(n)) \;, \; n \geq n_1$

$t_2(n) \leq c_2(g_2(n)), \; n \geq n_2$

$t_1(n) + t_2(n) \leq c_1(g_1(n)) + c_2(g_2($

$\leq 2c_3 \max(g_1(n), g$

$\leq \max\{2c_3(g_1(n)), 2c_3$

$\therefore \; t_1(n) + t_2(n) \in O(\max\{g_1(n)$

2 a. With neat diagram explain a
in designing and analyzing

COMPASS

M T W T F S S

Date :

* Understanding the problem
* Deciding on: Computational means, Exact vs. approximate problem solving data structure(s), Algorithm design techniques.
* Design an algorithm
* Prove Correctness
* Analyze the correctness
* Code the algorithm

(USE the Sentence "Unexpectedly, Ducks drive peacefully around Cottarls" - to remember these steps)

U	- Understand
D	- Decide
D	- Design
P	- Prove
A	- Analyze
C	- Code

"SUCCESS DOESN'T COME TO YOU, YOU'VE GOT TO GO TO IT"

Date :

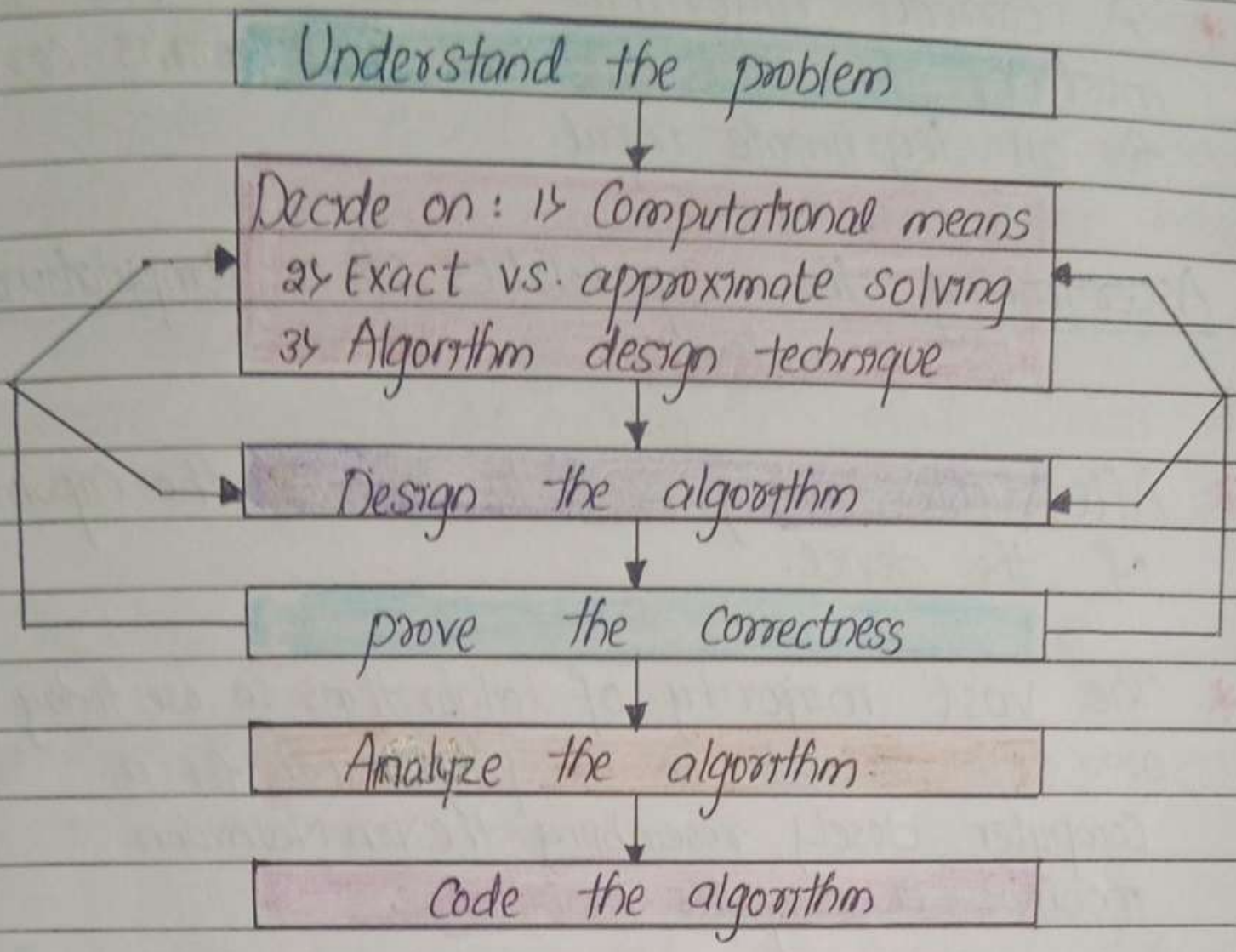


Figure: Algorithm design and analysis process.

Understanding the problem

- Before designing an algorithm the most important thing is to understand the problem given.
- Asking questions, doing a few examples by hand, thinking about special case, etc.
- An Input to an algorithm specifies an instance of the problem the algorithm that it solves.
- Important to specify exactly the range of instances the algorithm needs to handle else it will work correctly for majority of inputs but crash on some bandary values.

COMPASS

M T W T F S S

Date :

* A Correct algorithm is not one that works most of the time, but one that works correctly for all legitimate input.

Ascertaining the Capabilities of a Computational device

* After understanding need to ascertain the capabilities of the device.

* The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine - a computer architecture.

* Von Neumann machines are Sequential and the algorithms implemented on them are called Sequential algorithms.

* Algorithms designed to be executed on parallel computer are called parallel algorithms.

* For very Complex algorithms contentrate on a machine with high speed and more memory where time is critical.

Choosing between exact and approximate problem Solving

* For exact result -> exact algorithm

M T W T F S S

COMPASS

Date :

* for approximate result -> approximation algorithm

* Examples of exact algorithms: Obtaing square roots for numbers and solving non-linear equations.

* An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

Deciding on appropriate data structures

* Algorithms may or may not demand ingenuity in representing the inputs

* Inputs are represented using various data structures.

* Algorithm + data structures = program.

Algorithm Design Techniques and Methods of Specifying an Algorithm

* 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.

* The different algorithm design techniques are: brute force, approach, divide and conquer, greedy method, decrease and conquer, dynamic programming transform and conquer and back tracking.

M T W T F S S

COMPASS

Date :

proving an Algorithm's Correctness

* After specifying an algorithm we have to prove its correctness.

* The correctness is to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

* For example, the correctness of Euclid's algorithm for computing the greatest comman divisor stems from the correctness of the equality gcd(m,n) = gcd(n, m mod n), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

* For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.

* A common technique for proving correctness is to use "mathematical induction."

* Proof by mathematical induction is most appropriate for proving the correctness of an algorithm.

Analyzing an algorithm

* After correctness, efficiency has to be estimated.

M T W T F S S

Date :

→ Time efficiency and Space efficiency

→ Time : how fast the algorithm runs?

→ Space : how much extra memory the algorithm needs?

* Simplicity : how simpler it is compared to existing algorithms. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

* Generality : Generality of the problem the algorithm solves and the set of inputs it accepts.

* If not satisfied with these three properties it is necessary to redesign the algorithm.

Code the algorithm

* Writing program by using programming language.

* Selection of programming language should support the features mentioned in the design phase

* program testing : If the inputs to algorithms belong to the specified sets then require no verification. But while implementing algorithms as programs to be used in actual applications. it is required to provide such verifications.

b. Explain the general plan for
efficiency of a non-recursive
Suggest a non-recursive algo
maximum element in the list

M T W T F S S

COMPASS

Date :

Mathematical Analysis Of non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating the 'input size of the algorithm.'

2. Identify algorithm's 'basic operation'

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, then investigate 'worst, average and best case efficiency' seperately

4. Set up summation for C(n) reflecting the number of times the algorithm's basic operation is executed.

5. Simplify summation using standard formulas and express C(n) using order of growth

Example: Finding the largest element in a given array

```
ALGORITHM MaxElement(A[0...n-1])
// Determines the value of the largest element in given array
//Input: An Array A[0..n-1] of real number
//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

Analysis:

1. Input size: the number of elements = n (size of array)
2. Two operations can be considered to be as basic operation i.e,
   → Comparison : A[i] > maxval
   → Assignment : maxval ← A[i].
   Here the comparison statement is considered to be the basic operation of the algorithm.
3. No best, worst, average cases - because the number of comparisons will be same for all arrays of size n and it is not dependent on type of input.
4. Let C(n) denotes number of comparisons : Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bound between 1 and n-1.

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

This is an easy sum to compute because it is nothing other than 1 repeated n-1 times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 \quad 1+1+1+\cdots+1 \quad [(n-1) \text{ number of times}]$$

$$C(n) = \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = n-1 \qquad \sum_{i=l}^{u} 1 = u - l + 1$$

$$C(n) \in \Theta(n)$$

C. With the algorithm derive Case efficiency for Bubbl

BUBBLE SORT

The bubble sorting algorithm adjacent element of the list them if they are out of order repeatedly, we end up "bubbling up element to the last position o next pass bubbles up the secon element, and so on, until after the list is sorted pass i Co ≤ bubble sort can be represented following : $A_0, \ldots A_j \overset{?}{\leftrightarrow} A_{j+1} \ldots, A_{n-}$

M T W T F S S

COMPASS

Date :

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 non decreasing 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]

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as on example.

(89) ↔	(45)	68	90	29	34	17
45	(89) ↔	(68)	90	29	34	17
45	68	(89) ↔	(90) ↔	(29)	34	17
45	68	89	29	(90) ↔	(34)	17
45	68	89	29	34	(90) ↔	(17)
45	68	89	29	34	17	90
(45) ↔	(68) ↔	89 ↔	(29)	34	17	90
45	68	29	(89) ↔	(34)	17	90
45	68	29	34	(89) ↔	(17)	90
45	68	29	34	17	89	90

etc...

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for Selection Sort:

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

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

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

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is same as the number of key comparisons

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

MODULE-2

1. a. Explain the Concept of divide and Conque Design an algorithm for merge sort and derive its time Complexity

A divide and Conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer).

M T W T F S S

COMPASS

Date :

ANALYSIS AND DESIGN OF ALGORITHMS

Model Question Paper - 2023-24

MODULE-2

1. a. Explain the Concept of divide and Conquer Design an algorithm for merge sort and derive its time complexity

A divide and Conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer).

Date :

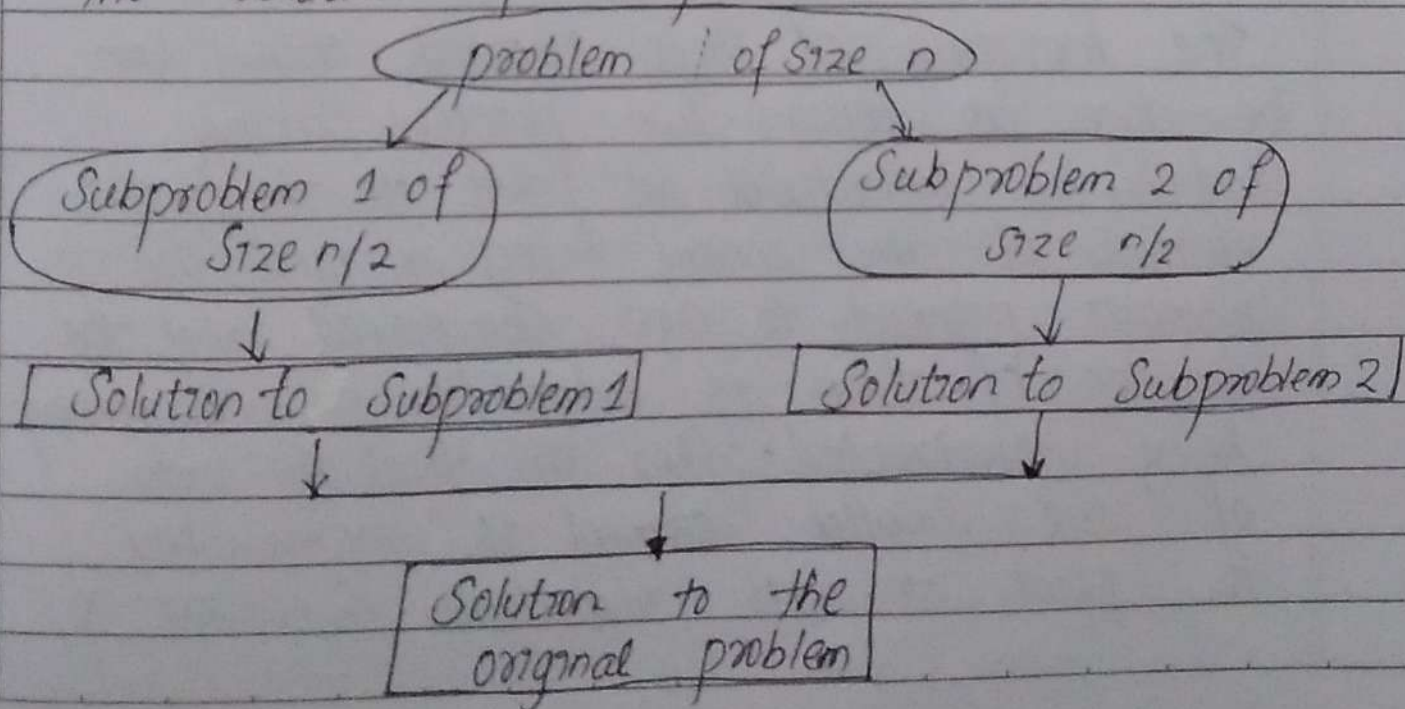
Divide-and-conquer algorithm work according to the following general plan:

1) A problem is divided into several subproblem 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 solution to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique as shown in Figure 2.4, which depicts the case of dividing a problem into two smaller subproblems, then the subproblems solved seperately. Finally solution to the original problem is done by combining the solution of subproblems.



MERGE SORT

Mergesort is based on 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.

ALGORITHM Mergesort (A[0..n-1])
// Sort array A[0..n-1] by recursive mergesort
// Input: An array A[0..n-1] of orderable elements
// Output: Array A[0..n-1] sorted in nondecreasing order

if n>1
    Copy A[0..⌊n/2⌋-1] to B[0..⌊n/2⌋-1]
    Copy A[⌊n/2⌋..n-1] to C[0..⌊n/2⌋-1]
    Mergesort (B[0..⌊n/2⌋-1])
    Mergesort (C[0..⌊n/2⌋-1])
    Mergesort (B,C,A) //see below

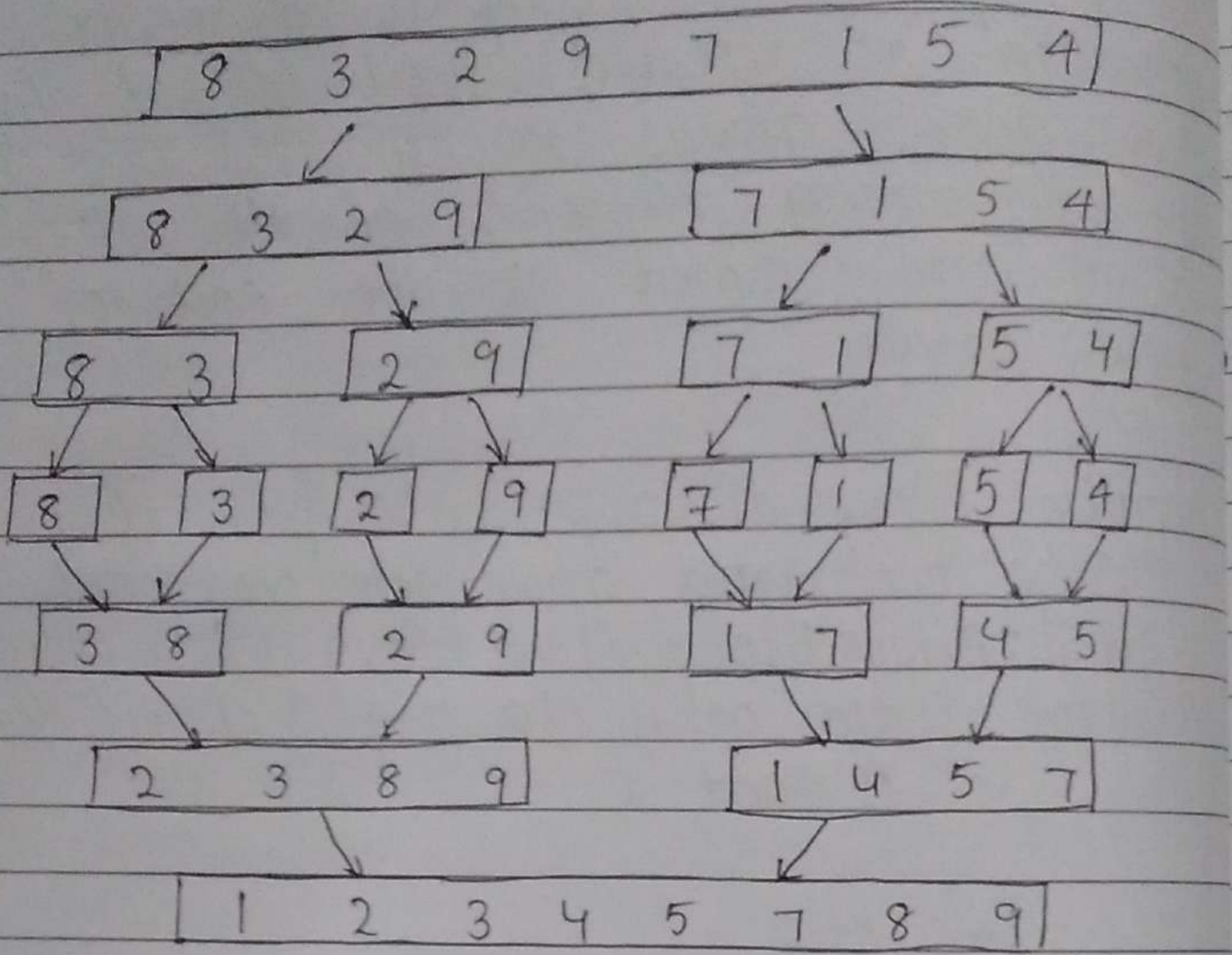
The merging of two sorted array can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the array being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after all that, the index of the smaller element is incremented to point to its immediate successor in

the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

Algorithm Mergesort(B[0...p-1], C[0..q-1], A[0..p+q-1])
//Merges two sorted arrays into one sorted array
// Input: Arrays B[0...p-1] and C[0..q-1] Both sorted
// Output: Sorted array A[0...p+q-1] of the element of B and C

```
i←0; j←0; k←0
while i<p and j<q do
    if B[i] ≤ C[j]
        A[k]←B[i]; i←i+1
    else A[k]←C[j]; j←j+1
    k←k+1
if i=p
    copy C[j..q-1] to A[k...p+q-1]
else copy B[i...p-1] to A[k..p+q-1]
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated below.



Example of mergesort operation.

The recurrence relation for the number of key comparisons $C(n)$ is

$C(n) = 2C(n/2) + C_{merge}(n)$ for $n>1$, $C(1)=0$.

In the worst case, $C_{merge}(n) = n-1$ and we have the recurrence.

$C_{worst}(n) = 2C_{worst}(n/2) + n-1$ for $n>1$, $C_{worst}(1)=0$

By Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$

the exact solution to the worst-case recurrence for $n=2^k$

$C_{worst}(n) = n\log_2 n - n + 1$.

COMPASS

Date :

For large n, the number of comparisons made by this algorithm in the average case turns out to be 0.25n less and hence is also in $O(n \log n)$

b. Design an insertion sort algorithm and obtain its time complexity. Apply insertion sort on these elements. 25, 75, 40, 10, 20

Insertion Sort:

Let us consider an application of the disease-by-one technique to sorting an Array A[0... n-1] following the technique's idea. we assume that the smaller problem of sorting the array A[0... n-2] has already been solved to given us a sorted array of size n-1; $A[0] \leq \dots \leq A[n-2]$.

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

```
//Sort a given array by insertion sort
// Input : An Array A[0... n-1] of n orderable elements
// Output : Array A[0..n-1] Sorted in nondecreasing order.
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

$A[0] \leq \dots \leq A[j] < A[j+1] \leq \dots \leq A[i-1] \mid A[i] \dots A[n-1]$

M T W T F S S

Date :

Time Complexity:

* The number of key comparisons in this algorithms obviously depends on the nature of the input.

* In the worst case, $A[j] > v$ is executed the largest number of times, i.e, for every $j = i-1, \ldots 0$. The worst-case input is an array of strictly decreasing values. The number of key comparisons for such as input is,

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

* In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \leq A[i]$ for every $i = 1, \ldots n-1$, i.e, if the input array is already sorted in nondecreasing order. Thus, for sorted arrays, the number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

* On randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing array i.e,

COMPASS

Date :

$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$

Given elements: 25, 75, 40, 10, 20

(25)	| (75)	40	10	20
25	(75)	(40)	10	20
25	40	(75)	(10)	20
10	25	40	(75)	(20)
10	20	25	40	75

The Sorted list of elements: 10, 20, 25, 40, 75

OR

4a) Explain Strassen's matrix multiplication and derive its time Complexity.

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

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

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

where,

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

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

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

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

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

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

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

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes 7 multiplication and 18 additions /Subtraction, whereas the brute-force algorithm require 8 multiplications and 4 additions. These numbers should not lead us to multiplying 2×2 matrices by Strassen's algorithm. Its importance stems from its asymptotic superiority as matrix order n goes to infinity.

Let A and B be two n×n matrices where n is a power of 2. (If n is not power of 2, matrices can be padded with rows and colums of zeros). We can divide A, B and their product C into four n/2 x n/2 Submatrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

The value of C_{00} be computed as either $A_{00} * B_{00} + A_{01} * B_{10}$ or as $m_1 + m_4 - m_5 + m_7$ where

COMPASS

Date :

M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. The seven products of $n/2 \times n/2$ matrices are computed recursively by Strassen's matrix multiplication algorithm.

The asymptotic efficiency of Strassen's matrix multiplication algorithm

If $M(n)$ is the number of multiplication made by Strassen's algorithm in multiplying two $n \times n$ matrices, where n is a power of 2. The recurrence relation is $m(n) = 7m(n/2)$ for $n>1$, $m(1)=1$

Since $n = 2^k$

$$m(2^k) = 7m(2^{k-1})$$
$$= 7[7m(2^{k-2})]$$
$$= 7^2 m(2^{k-2})$$
$$\cdots$$
$$= 7^i m(2^{k-i})$$
$$= \cdots$$
$$= 7^k m(2^{k-k}) = 7^k m(2^0) = 7^k m(1)$$
$$= 7^k$$

$$m(2^k) = 7^k$$

Since $k = \log_2 n$,

$$m(n) = 7^{\log_2 n}$$
$$= n^{\log_2 7}$$
$$\approx n^{2.807}$$

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

b. Design an algorithm for quick sort algorithm. Apply quick sort on these elements.
25, 75, 40, 10, 20, 05, 15

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. quicksort divide input elements according to their value. A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

A[0] .... A[s-1] | A[s] | A[s+1] ... A[n-1]

all are ≤ A[s] | all are ≥ A[s]

Sort the two subarrays to the left and to the right of A[s] independently. No work required to combine the solution to the subproblems.
Here is pseudocode of quicksort: Call Quicksort(A[0..n-1]) where As a partition algorithm use the HoarePartition.

ALGORITHM Quicksort(A[l..r])
//Sorts a subarray by quicksort
// Input: Subarray of array A[0..n-1], defined by
// its left and right indices l & r

M T W T F S S

COMPASS

Date :

// output : Subarray A[l..r] sorted in nondecreasing order
if l<r
    s ← HoarePartition (A[l..r])
    Quicksort (A[l..s-1])
    Quicksort (A[s+1..r])

Given list : 25, 75, 40, 10, 20, 05, 15

	0	1	2	3	4	5	6
	p	i					j
Solution:-	25	75	40	10	20	05	15
pivot = 25							
	25	15	40	10	20	05	75
	25	15	05	10	20	40	75
	20	15	05	10	25	40	75

Sorting left sub array

	0	1	2	3	4
p	20	15	05	10	25
	20	15	10	05	25
	05	15	10	20	25

Combining left and right subarray

05 15 10 20 25 40 75

Condition and rules

* if p>i, increment i
* if p<j, increment j
* if p≯i, stop increment and compare with j
* if p≮j, stop increment and compare with i
* if index of i is less than index of j Swap i,j
* If index of j is lesser Swap pivot, j

M T W T F S S

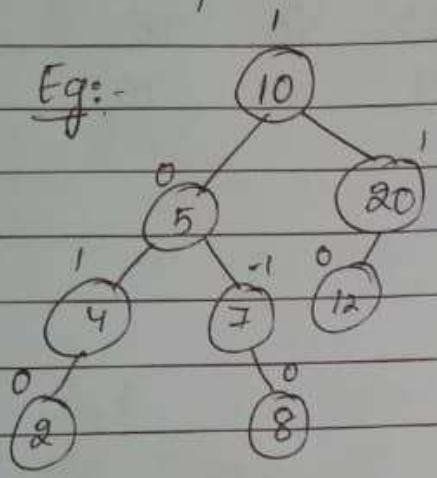
Date :

MODULE-3

5 a) Define AVL Trees. Explain its four rotation types.

AVL trees is a binary search tree in which the balance factor of every node, which is defined as the difference between the height of node's left and right subtrees, is either 0 or +1 or -1. (The height of the empty tree is defined as -1).

Eg:-

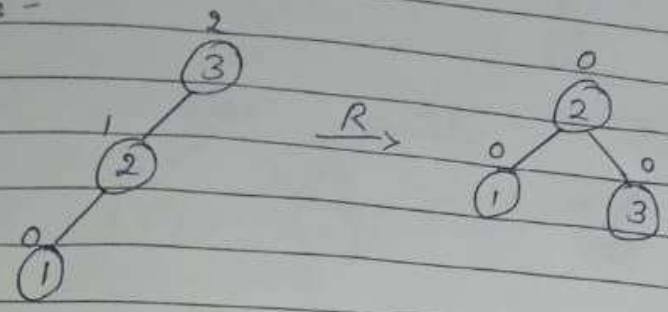


1. Single right rotation: or R-rotation

* Imagine rotating the edge connecting the root and its left child in the binary tree in the right.
* Single R-rotation is general form.
* This rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion

COMPASS
Date :

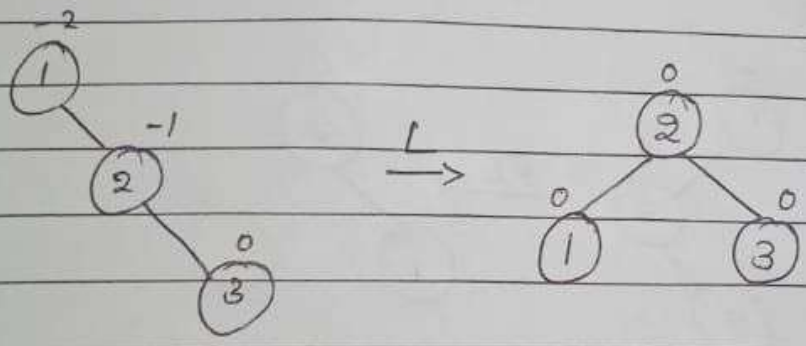
Eg:-



2. Single left rotation or L-rotation.

* The mirror image of the single R-rotation.
* It is performed after the new key is inserted into the right subtree of the right child of the tree whose roots had been balanced by -1 before the insertion.

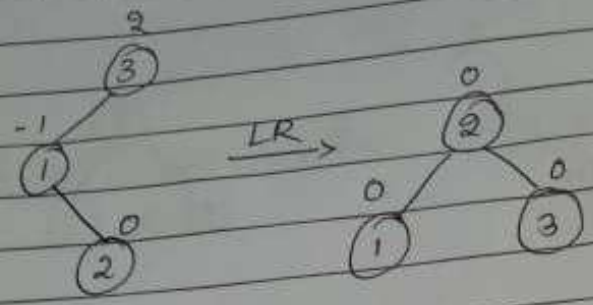
Eg:-



3. Double left-right rotation or LR rotation:

* A Combination of two rotations.
* We perform L-rotation of the left subtree of root r followed by the R-rotation of the new tree rooted at r.
* It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.

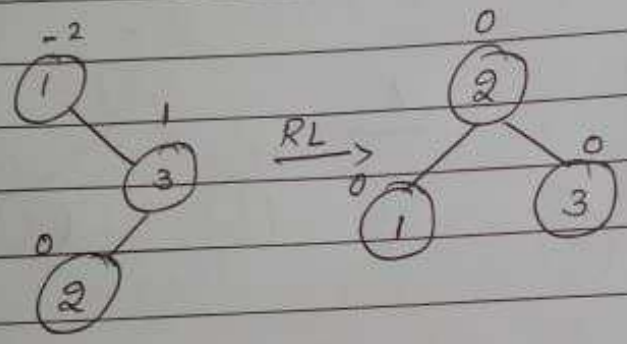
Eg:-



4. Double right-left rotation or RL rotation:

* The mirror image of double left-right rotation.
* It's Combination of two rotations:
  1. A 90-degree rotation to the right
  2. A 90-degree rotation to left.

Eg:-



6) Construct bottom up heap for the list 2,9,7,6,5,8. obtain its time Complexity

Bottom-up heap construction:

It initalizes the essentially complete binary tree with n nodes by placing keys in the order given and then 'heapfiles' the tree as follows

COMPASS

Date :

*Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key at this node.

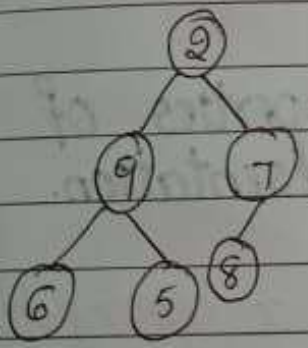
*If it does not, the algorithm exchanges the node's key K with the larger key of its children and checks whether the parental dominance hold for K in its new position. This process continues until the parental dominance requirement for K is satisfied.

* After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor.

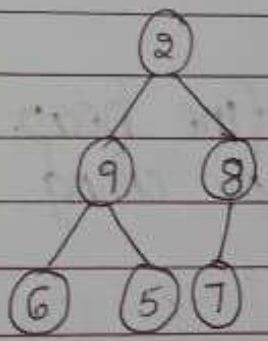
Given list: 2, 9, 7, 6, 5, 8

2, 9, 7, 6, 5, 8

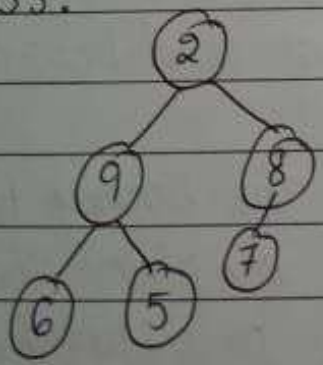
S1: Construct binary tree.



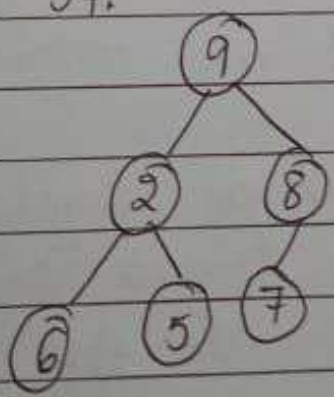
S2: Comparison starts



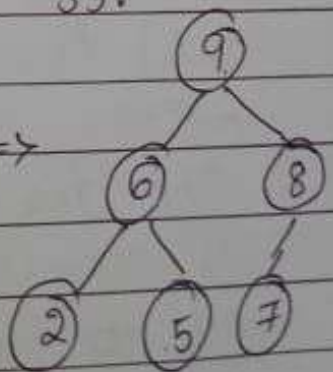
S3:



S4:



S5:



Constructed heap: 9, 6, 8, 2, 5, 7

COMPASS

Date :

M T W T F S S

Time Complexity :-

Let $n = 2^k - 1$

h = height of tree $\Rightarrow h = [\log_2 n]$

* Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithms.
* moving next level down need two comparisons - one to find the larger child and the other to determine whether the exchange is required
* Total number of key comparisons involving a key on level i will be $2(h-i)$, key comparison in worst case: $C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ key}} 2(h-i) = \sum_{i=0}^{h-1} 2^i 2(h-i)$

$= 2(n - \log_2(h+1))$.

OR

6) a) Define heap. Explain the properties of heap along with its representation.

A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

1. The tree's shape requirement - the binary tree is essentially complete (or simply complete), that is all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

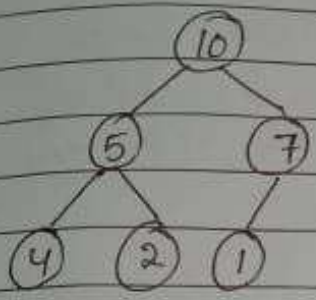
M T W T F S S

COMPASS

Date :

2. The parental dominance requirement - the key at each node is greater than or equal to the keys at its children.

Heap Example



Index	0	1	2	3	4	5	6
Value		10	5	7	4	2	1
		parents			leaves		

array representation

Here is a list of important properties of heap

1. There exists exactly one essentially complete binary tree with n nodes. It's height is equal to $\log_2 n$.
2. The root of a heap always contains its largest element
3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 though n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,

a. the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lfloor n/2 \rfloor$ positions.

b. the children of a key in the array's parental position i ($1 \le i \le \lfloor n/2 \rfloor$) will be in positions $2i$ and $2i+1$, and correspondingly, the parent of a key in position i ($2 \le i \le n$) will be in position $\lfloor i/2 \rfloor$.

$H[i] \ge \max\{H[2i], H[2i+1]\}$ for $i = 1, \ldots, \lfloor n/2 \rfloor$.

b. Design Horspools algorithm for string matching. Apply Horspools algorithm to Find the pattern BARBER in the text: JIM_SAW_ME_IN_A_BARBERSHOP

Algorithm:-

```
ShiftTable(P[0.... m-1])
i ← m-1
while i ≤ n-1 do
    k ← 0
    while k ≤ m-1 and P[m-1-k] = T[i-k] do
        k ← k+1
    if k = m
        return i-m+1
    else i ← i + Table[T[i]]
return -1
```

M T W T F S S

COMPASS

Date :

Given text: JIM_SAW_ME_IN_A_BARBERSHOP

pattern: BARBER

JIM_SAW_ME_IN_A_BARBERSHOP

BARBER

(shift by 4 letters) BARBER

(shift by 1 letter) BARBER

Cas - R not present shift complete length) BARBER

(after B two letters one present shift at by 2 letters) BARBER

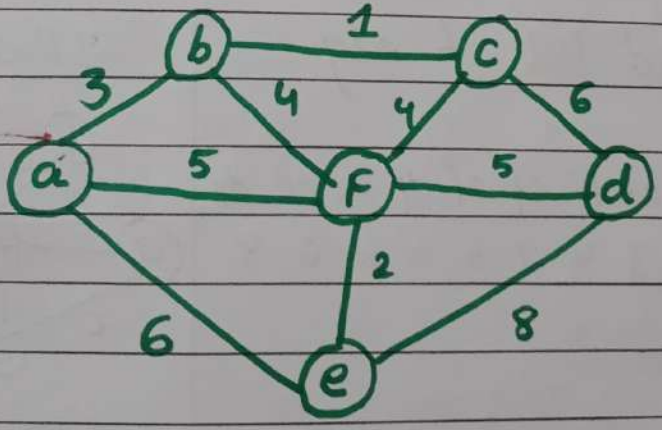
BARBER. Complete pattern is matched.

BARBERSHOP

JIM SAW ME IN A BARBERSHOP
BARBER
(shift by 4 letters) BARBER
(shift by 1 letter) BARBER.
(as - 'r' not present shift complete length) BARBER
(after B two letter are present shift by 2 letters) BARBER
BARBER. Complete pattern is matched.

MODULE-4

7a) Construct minimum cost spanning tree using kruskals algorithm for the following graph



ALGORITHM Kruskal(G)
// Kruskal's algorithm for constructing a minimum
// Spanning tree
// Input: A weighted connected graph $G=\{V,E\}$
// Output: E_T, the set of edges composing a
// minimum spanning tree of G Sort E in
non-decreasing order of the edge weight $w(e_{i_1}) \leq \ldots \leq w(e_{i_{|E|}})$

Date :

M T W T F S S

$E_T \leftarrow \phi$; ecounter $\leftarrow 0$ //initialize the set of tree edge and its size

$k \leftarrow 0$

while ecounter $< |v| - 1$ do

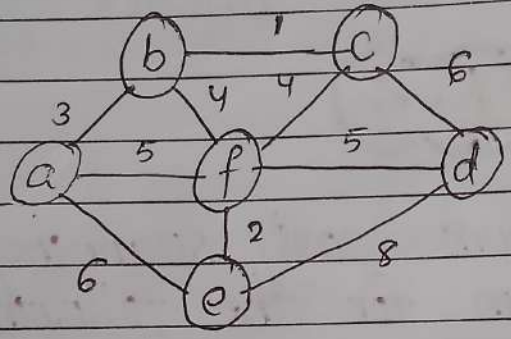
$k \leftarrow k+1$

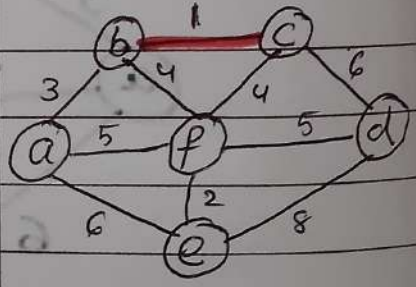
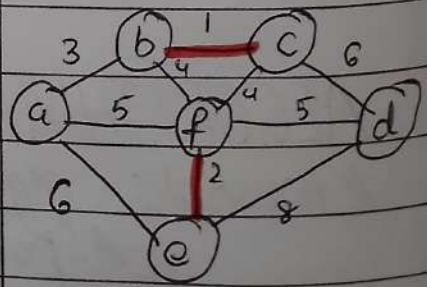
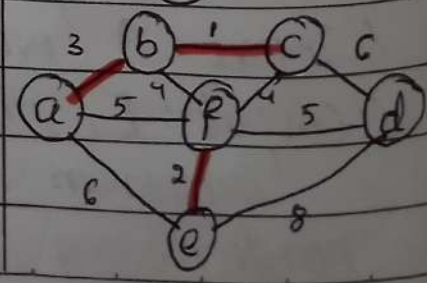
if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ik}\}$; ecounter $\leftarrow$ ecounter $+1$

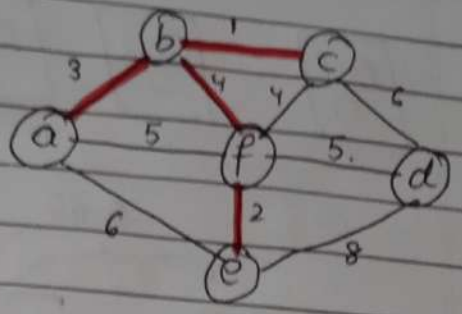
return E_T

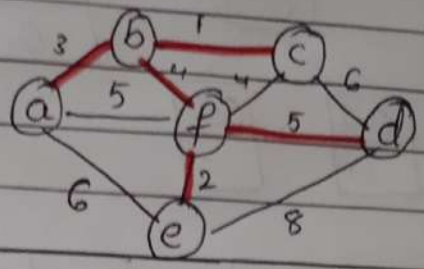
Given graph:-



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

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





b) What are Huffman Tree? Construct Huffmann tree for the following data.

character	A	B	C	D	E	-
probability	0·5	0·35	0·5	0·1	0·4	0·2

Encode DAD-CBE using Huffman Encoding

Given:-

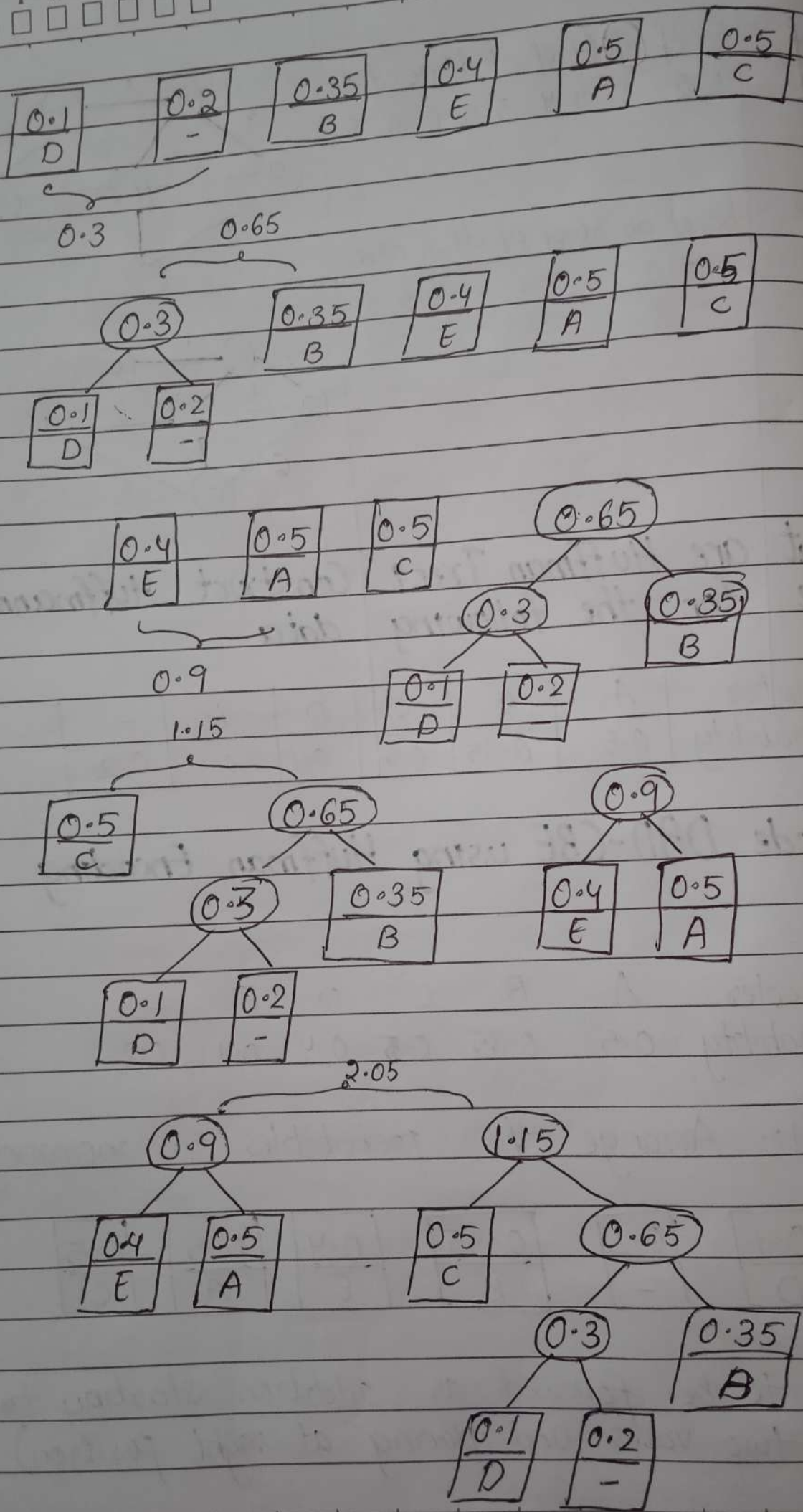
character	A	B	C	D	E	-
probability	0·5	0·35	0·5	0·1	0·4	0·2

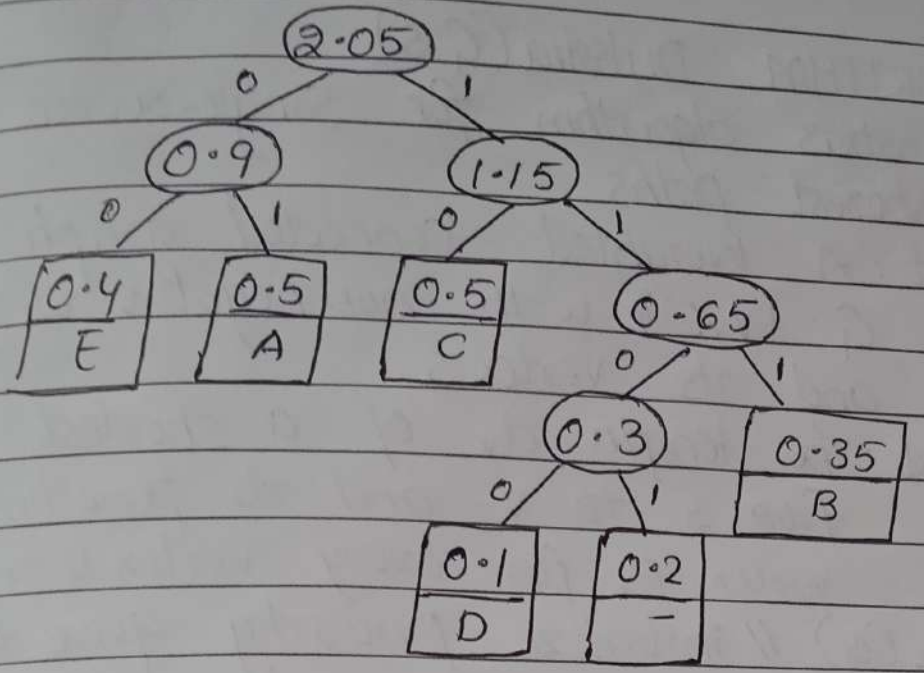
Step 1:- Arrange the probabilities in increasing order.

0·1	0·2	0·35	0·4	0·5	0·5
D	-	B	E	A	C

(Step to be followed is addition starting two two values and placing at right position)

M T W T F S S



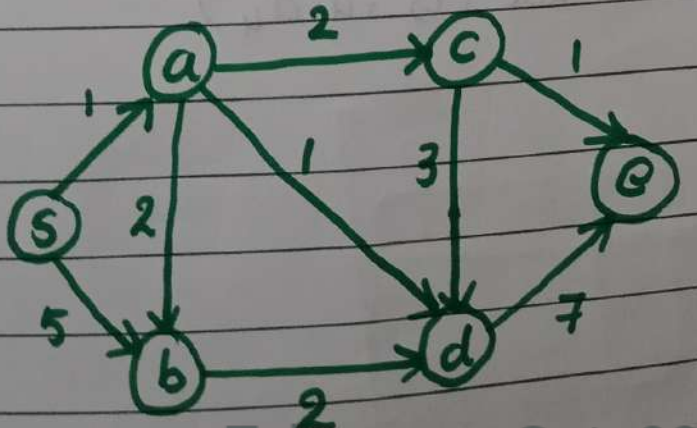


Character	A	B	C	D	E	-
probability	0·5	0·35	0·5	0·1	0·4	0·2
Codeword	01	111	10	1100	00	1101

DAD_CBE is encoded as 11000111001101101100

OR

8) a. Apply Dijkstra's algorithm to find Single source shorest path for the given ~~path~~ graph by Considering S as the source vertex!



M T W T F S S

ALGORITHM Dijkstra(G, s)
// Dijkstra's algorithm for Single-source Shortest paths
// Input: A weighted connected graph $G = \{V, E\}$ with non-negative weights and its vertex s
// output: The length d_v of a shortest path from s to v and its penultimate vertex p_v for every vertex v in V
Initialize(Q) // initialize priority queue to empty
for every vertex v in V
    $d_v \leftarrow \infty$; $p_v \leftarrow$ null
    Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; Decrease(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \phi$
for i $\leftarrow$ 0 to |V| - 1 do
    $u^* \leftarrow$ DeleteMin(Q) //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - V_T$ that is adjacent to u^* do
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
            Decrease(Q, u, d_u)

b. Define transitive closure of a graph. Apply Warshalls algorithm to compute transitive closure of a directed graph.

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Definition:- The transitive closure of a directed graph with n vertices can be defined as the n-by-n boolean matrix $T = \{t_{ij}\}$, in which the element in the i^{th} row ($1 \le i \le n$) and the j^{th} column ($1 \le j \le n$) is 1 if there exists a nontrivial directed path (i.e., a directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.

$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

COMPASS

Date :

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)}$ =

a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

MODULE-5

Q) a) Explain the following with examples

i) P problem

ii) NP problem

iii) NP-Complete problem

iv) NP-Hard problem

R =

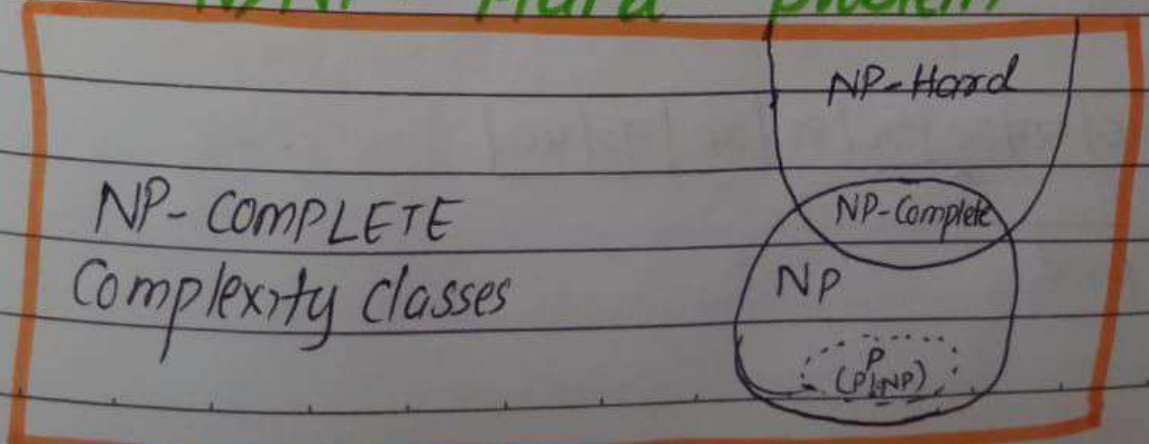
	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)}$ =

a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

MODULE-5

9) a) Explain the following with examples
   i) P problem
   ii) NP problem
   iii) NP-Complete problem
   iv) NP-Hard problem



M T W T F S S

Date:

P problem: class P is a class of decision problem that can be solved in polynomial time by (deterministic) algorithms. This class of problem is called polynomial.

Or

pdynomial time problems, commonly known as P problem, the solution of the problem can be found in polynomial time.

Example:- Linear Search,

Find the element 30 in the array
a[] = {10, 50, 30, 70, 80, 20, 90, 40}

arr[]	10	50	30	70	80	20	90	40

Key = 30

S-1

30

Key

10	50	30	70	80	20	90	40

Not equal

Key not found.

S-2

30

10	50	30	70	80	20	90	40

Not equal

Key not found

S-3

30

10	50	30	70	80	20	90	40

equal

Key found

NP Problem:- Nondeterministic polynomial-tree problems, commonly known as NP problem. these problems have the special property that, once a potential solution is provided, its correctness can be verified quickly. However, finding the solution itself may be computationally difficult

or

Class NP is the class of decision problem that can be solved by nondeterministic polynomial algorithm. This class of problems is called nondeterministic polynomial.

Example: prime factorization.

pseudocode

```
primefactors[] // to store result
i=0 // Index of primefactors
while n!=1;
//SPF: Smallest prime factor
primefactors[i] = SPF[n]
i++
n= n/SPF[n]
```

Time Complexity:- O(log n), for each query (Time Complexity for percomputation is not included)

COMPASS

M T W T F S S

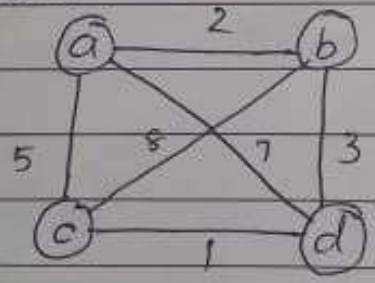
Date :

NP-Complete problem: A decision problem D_1 is said to be polynomially reducible to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that

1. t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instance of D2;
2. t is computable by a polynomial-time algorithm.

Example:- Traveling salesman problem

This problem asks to find the shortest tour through a given set of n cities that visit each city exactly once before returning its starting position.



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

M T W T F S S

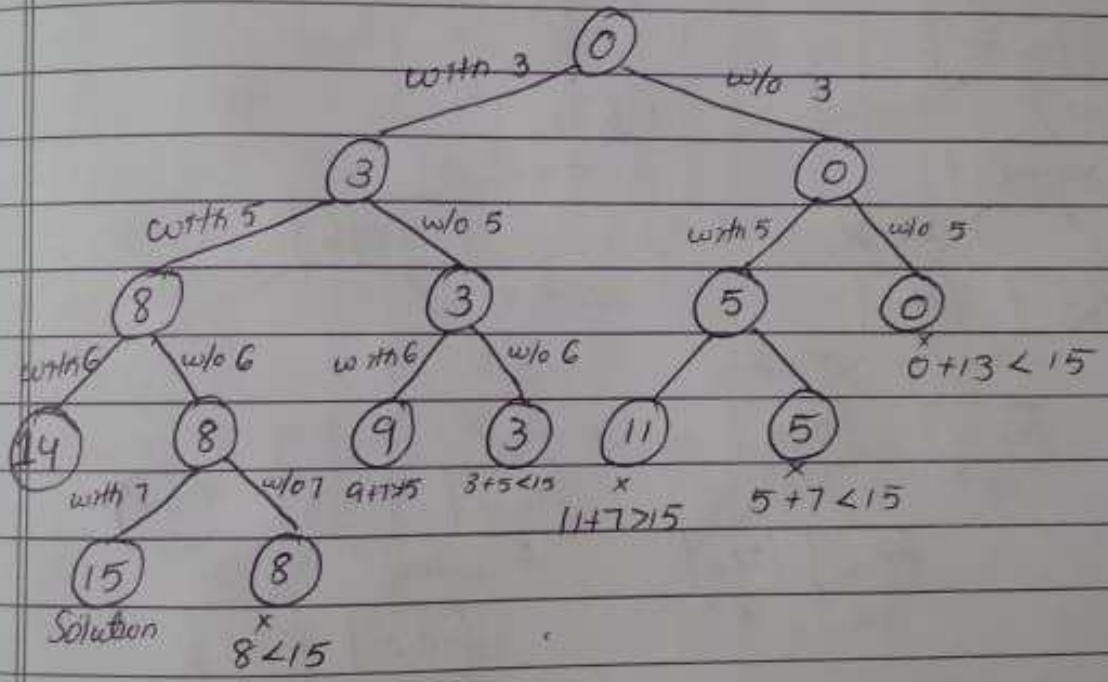
COMPASS

Date :

NP-Hard problem: A 'P' problem is said to be NP-Hard when all 'Q' belonging to NP can be reduced in polynomial time (cn^k where k is some constant) to 'P' assuming a solution for 'P' takes 1 unit time.

Example:- Subset Sum problem

$S = \{3, 5, 6, 7\}$ and $d = 15$, we apply backtracking algorithm to find the subset-sum problem.



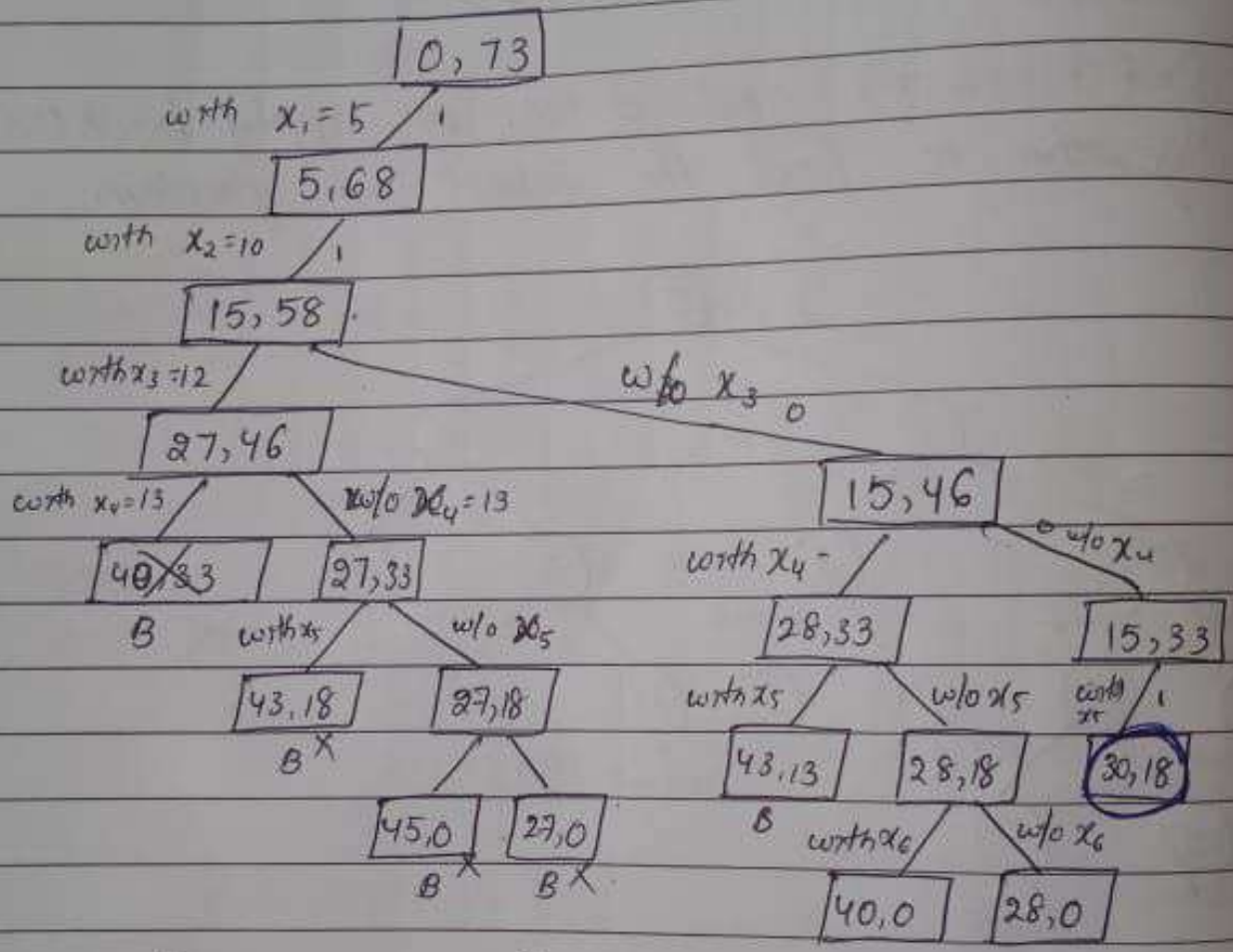
b) What is backtracking? Apply backtracking to solve the below instance of sum of subset problem
$S = \{5, 10, 12, 13, 15, 18\}$ $d = 30$

Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they

backtrack and try another unit they find the right one. It's like solving a puzzle by testing different pieces until they fit together perfectly

Given:- $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ $\{5, 10, 12, 13, 15, 18\}$

$n=6$, $m=30$



	5	10	12	13	15	18
x	1	1	0	0	1	0
	1	2	3	4	5	6

∴ the Subset is $\{5, 10, 15\}$

OR

10 a) Illustrate N queen's problem using backtracking to solve 4-Queens problem

M T W T F S S

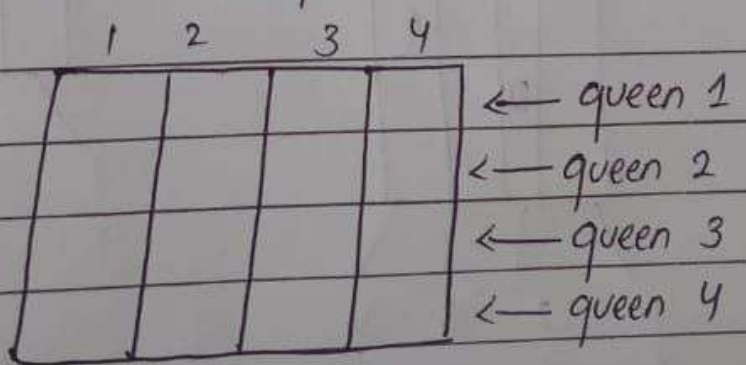
COMPASS

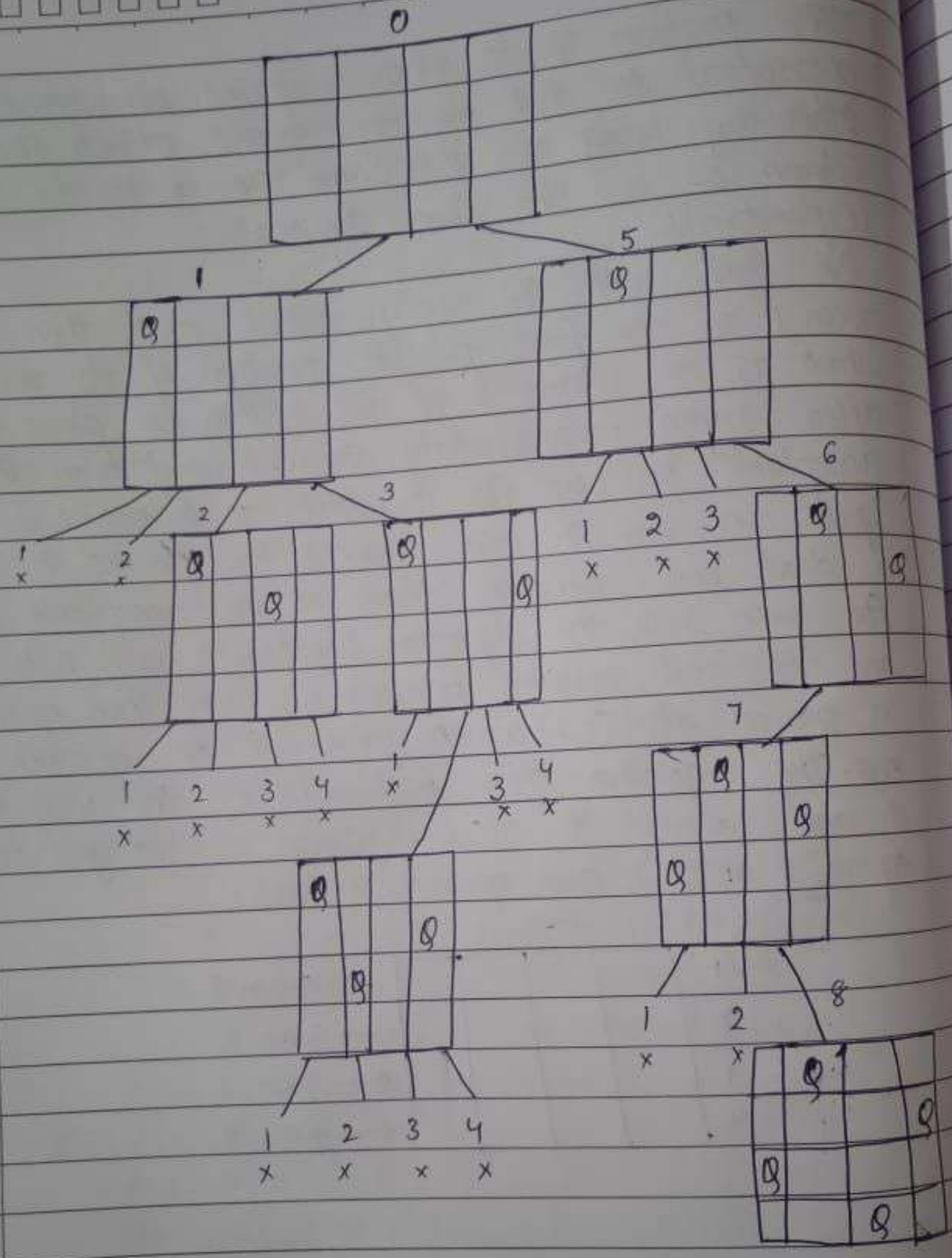
Date :

The problem is to place n queens on an n-by-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.

(Explanation)↓.

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 column 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).





State-space tree of solving the four-queen problem by backtracking.

M T W T F S S

COMPASS

Date :

b) Using Brach and Bound technique solve the below instance of knapsack problem

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	5

Capacity 5

Item	Weight	value	value/weight
1	2	12	6
2	1	10	10
3	3	20	6.666
4	2	5	2.5

$ub = V + (W-w)(V_{i+1}/W_{i+1})$

Node 0, $i=0$, $w=0$, $V=0$, $W=5$

$(V_{i+1}/W_{i+1}) = 6$

$ub = V + (W-w)(V_{i+1}/W_{i+1})$

$= 0 + (5-0)(6)$

$\boxed{ub = 30}$

Node 1, $i=1$, $w=2$, $V=12$, $V_{i+1}/W_{i+1} = 10$, $W=5$

$ub = V + (W-w)(V_{i+1}/W_{i+1})$

$= 12 + (5-2)(10)$

$= 12 + 30$

$\boxed{ub = 42}$

M T W T F S S

Date :

Node 2,

$i=2,\ w=1,\ V=10,\ V_{i+1}/W_{i+1}=6.66,\ W=5$

$ub = V + (W-w)(V_{i+1}/W_{i+1})$

$= 10 + (5-1)(6.66)$

$= 10 + 26.64$

$\boxed{ub = 36.64}$

Node 3,

$i=3,\ w=3,\ V=20,\ V_{i+1}/W_{i+1}=2.5;\ W=5$

$ub = V + (W-w)(V_{i+1}/W_{i+1})$

$= 20 + (5-3)(2.5)$

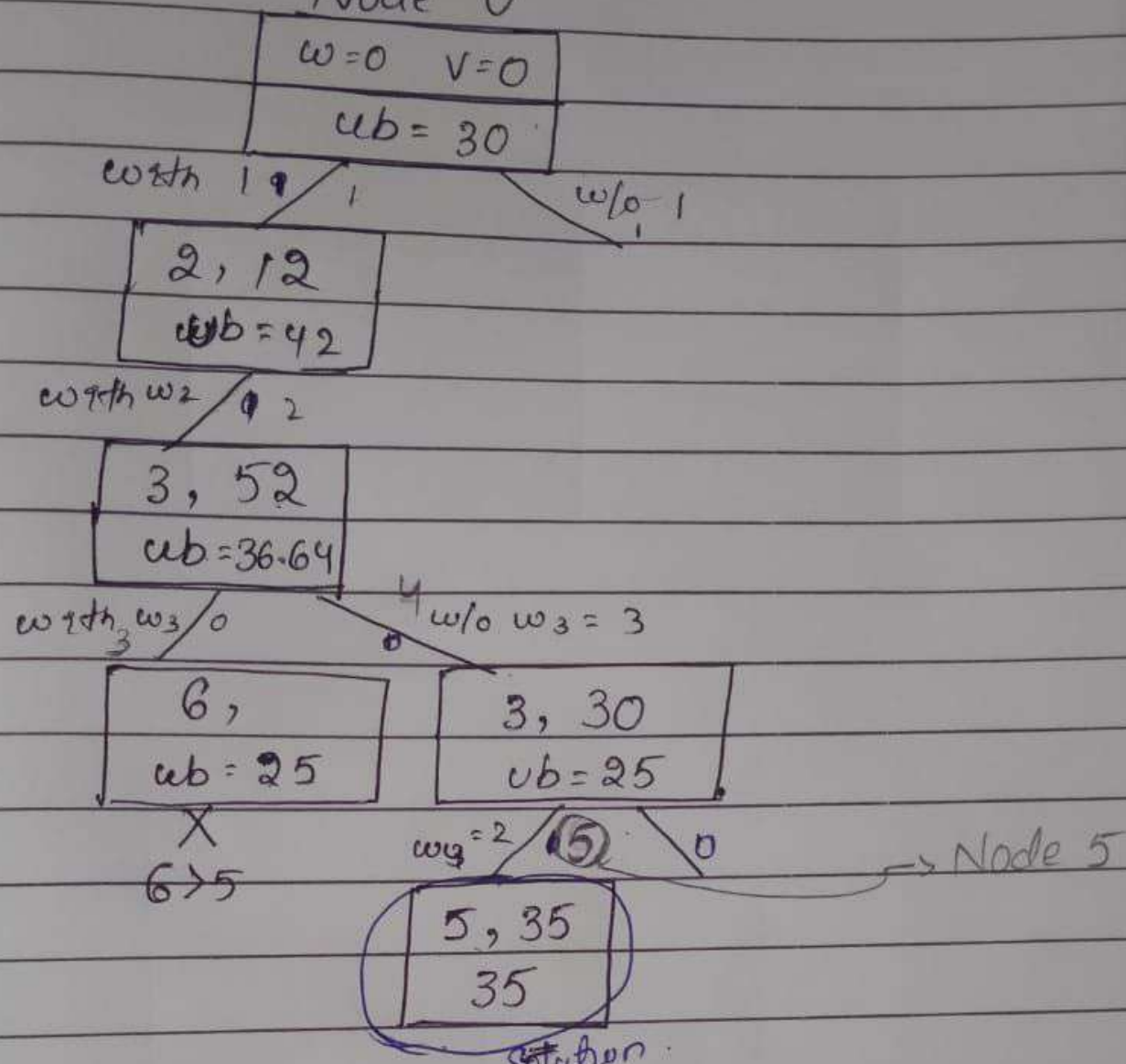
$= 20 + 5$

$\boxed{ub = 25}$

Node 5,

$ub = 35 + (5-5)(7)$

$\boxed{ub = 35}$



w_1	w_2	w_3	w_5
1	2	3	4
1	1	0	1
weights: 2	1	3	2

The Subset is {1, 2, 2}