VTU Solution - July 2025
SOLUTIONS
Question Paper: https://drive.google.com/file/d/193wL_7edc35GvQnJdf_bZGXOOdQmQRkL/view

Sub:	Analysis and Design of Algorithms	Sub Code:	BCS401	Branch:	CSE

		MARKS	CO	RBT
1				

		Module – 1	M	L	C
Q.1	a.	Define algorithm Explain asymptotic notations Bigh oh, Big omega and Big theta notations.	08	L2	CO1
	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.	08	L3	CO1
	c.	If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then show that $t_1(n) + t_2(n) \in o(\max\{g_1(n), g_2(n)\})$	04	L2	CO1

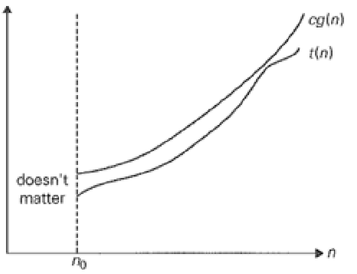
Solutions:

a. **Definition】An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:**
(1) Input There are zero or more quantities that are externally supplied.
(2) Output At least one quantity is produced.
(3) Definiteness Each instruction is clear and unambiguous.
(4) Finiteness If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after finite number of steps.
(5) Effectiveness Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in(3); it also must be feasible.

O-notation

DEFINITION 1 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 by some 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

$$t(n) \le cg(n) \quad \text{for all } n \ge n_0.$$



- **upper bound** of an algorithm's running time.
- Measures the **worst case time complexity** or longest amount of time an algorithm can possibly take to complete

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

Ω-notation

DEFINITION 2 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

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

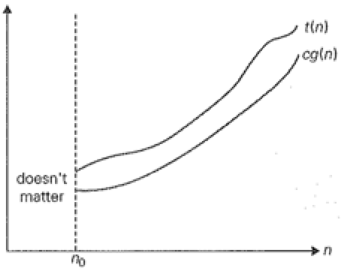


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

- **lower bound** of an algorithm's running time.
- It measures the **best case time complexity** or best amount of time an algorithm can possibly take to complete

Θ-notation

DEFINITION 3 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 constant c_1 and c_2 and some nonnegative integer n_0 such that

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

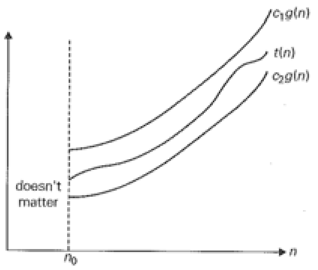


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

- **express both the lower bound and upper bound of an** algorithm's running time.
- **Average Case**

B. **General plan for Analyzing Time 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.

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

$F(n)$ is computed according to the formula

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:
if n = 0 return 1.

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

- For solving recurrence relations, We use method of backward substitutions.

$$M(n) = M(n-1) + 1 \qquad \text{substitute } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 = M(n-2) + 2 \qquad \text{substitute } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 = M(n-3) + 3.$$

$$M(n) = M(n-i) + i.$$

Since it is specified for $n = 0$, we have to substitute $i = n$

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

C.

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF (As you will see, the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, and b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ∎

Q.2	a.	With a neat diagram explain different steps in designing and analyzing algorithm.	08	L2	CO1
	b.	Write an algorithm to find the max element in an array of n elements. Give the mathematical analysis of this non- recursive algorithm.	08	L3	CO1
	c.	With the algorithm derive the worst case efficiency for selection sort.	04	L3	CO1

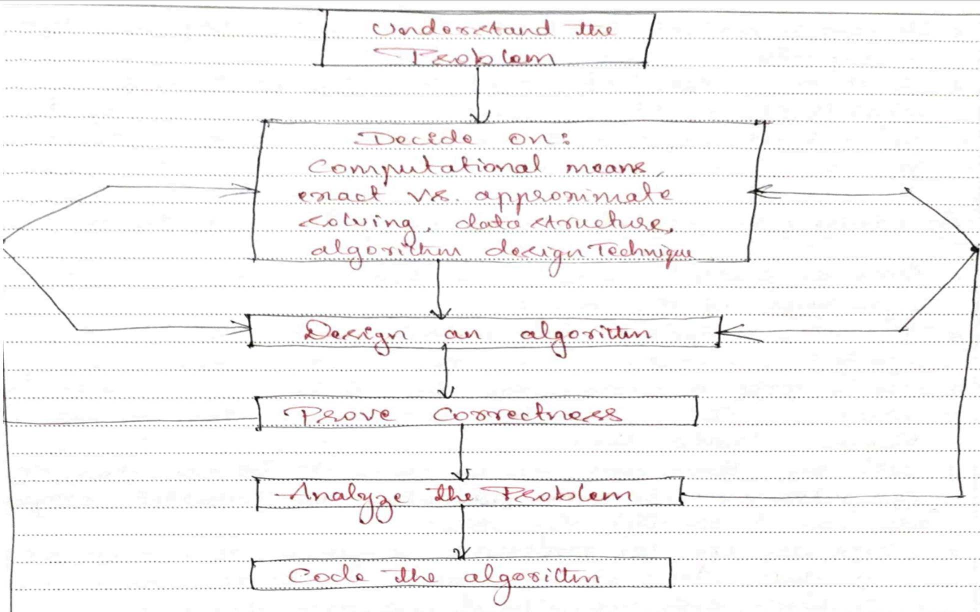
2.a.

Fundamentals of Algorithmic Problem Solving

* Algorithms are procedural solution to problems. These solutions are not answers but specific instructions for getting desired answers.

* The following daigram illustrates the sequence of steps involved in designing and analyzing an algorithm.

1) Understand the Problem
2) Ascertaining the Capabilities of a Computational device
3) Choosing between Exact and Approximate Problem solving
4) Deciding on Appropriate Data Structures
5) Algorithm Design Techniques
6) Methods of Specifying algorithm
7) Proving an Algorithms Correctness
8) Analyzing an algorithm
9) Coding an algorithm



1) Understanding problem

* Before the design, first thing you need to understand the complete problem given.
* Read problem's description and ask questions, by thinking special cases
* An input specifies an instance of the problem, the algorithm solves.

2) Ascertaining the capabilities of Computational device

* Once the problem understood clearly, we need ascertain the capabilities of the device.
* Because, computers that we use mostly are Von Neumann architecture based, where instructions are executed sequentially
* So we have to design algorithm in such a way that these algorithms should be executed and they are known as ~~sequent~~ Sequential algorithms
* The algorithms that are designed to be executed on parallel computers, which takes advantage of parallel computation are called parallel algorithms.
* There are complex problems, required to operate or manipulate huge data. In such situations we should concentrate on a machine with high speed and more memory

3) Choosing between Exact and Approximate Problem Solving

* The next principle decision is to choose between solving exactly or approximately. In the first case an algorithm is called an exact algorithm, second case it is an approximation algorithm.

* Why would one opt for an approximation algorithm?
  - → There are important algorithm problems that cannot be solved for most of their instances
    Eg: finding exact square root
    solving non-linear equation.
  - → available algorithms for solving a problem exactly can be unacceptably slow because of the problem intrinsic complexity.
  - → it can be a part of a more sophisticated algorithm that solves problem exactly.

) Deciding on Appropriate Data Structures

* Once the type of the algorithm, then we have to choose the proper data structures to represent the various data types or for easy manipulation so as to achieve the result.

Algorithms + Data structures = Programs.

) Algorithm Design Techniques

* An algorithm design technique, is a general approach to solve problems algorithmically that is applicable to a variety of problem
* Learning these techniques, is of utmost important for the following reasons.

  a) Provide the guidance for designing algorithms for new problem.
  b) make it possible to classify algorithms according to design idea.

Methods of Specifying an algorithm

* There are various methods of specifying an algorithm. algorithms can be expressed using.
  1) Natural language
  2) Pseudocode
  3) Flowchart.

7) Proof for algorithm correctness

* After specifying the algorithm for a specific problem we have to prove its correctness. It is responsibility to prove that algorithm will produces the required output for every legitimate input in a finite amount of time.
* mathematical induction is most appropriate for proving the correctness of the algorithm.

8) Analysis of algorithms

* Analysis indicates estimating time and space for a given problem.
* Once estimations is done, Select an algorithm which is more efficient interms of space & time.
* Time efficiency indicates, how fast the algorithm runs and space efficiency indicates, how much memory is required for the algorithm.
* Algorithm helps in identifying, which portion of the program consumes more time, so that it can be re-designed.

9) Implementation (coding an algorithm)

* The designed algorithm must be implemented using a programming language, such as C or C++ etc.

* The usage of different languages for solving a problem results in different memory requirements and affect the speed of the program.

* The selection of the programming language is important inorder to support the features mentioned in the design phase.

10) Program testing

* After implementing the algorithm in specific language next is the time to execute.

* After execution the desired result should be obtained. Testing is the verification of the code for correctness.

11) Documentation

* Documentation should exist from understanding the problem till it is tested and debugged. To understand design or code, proper comments should be given.

* Documentation enables the individuals to understand the programs written by other people.

So by definition we can write

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

where $c = 2c_3 = 2\max\{c_1, c_2\}$ $n_0 = \max(n_1, n_2)$

Mathematical analysis of Non-Recursive and Recursive algorithms

Analysis of Non-Recursive algorithm

General procedure:

1) Based on the input size, decide the various parameters to be considered.
2) Identify the basic operation to be performed in the innermost loop.
3) Compute the number of times the basic operation is executed and check whether the result obtained depends only in the size of the input. If the basic operation to be executed depends on some other conditions, then it is necessary to obtain the worst case, best case, average case seperately.
4) Obtain the total number of times basic operation is executed.
5) Simplify using standard formulas, compute the order of growth and express using asymptotic notations.

Algorithm to find the largest of n elements

Algorithm Maxelement (a[ ], n)
// Description: algorithm finds the largest element among array a[0 - - - n-1]
// Inputs: a[ ]: array elements
n: number of elements
// output: Maximum element from an array.

Step 1: big ← a[0]

Step 2: for i ← 1 to n-1 do
if (a[i] > big) then
big ← a[i]
end if
end for

Step 3: return big.

2.c.

ALGORITHM: Selection Sort (A [0 ... n-1])
// Sorts a given array by selection sort.
// Input: An array A [0 ... n-1] of orderable elements
// Output: An array A [0 ... n-1] sorted in ascending order.

```
for i ← 0 to n-2 do
    min ← A[i]
    for j ← i+1 to n-1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

Module – 2

Q.3	a.	Explain the concept of divide and conquer. Design an algorithm for merge sort and derive its time complexity.	10	L3	CO2
	b.	Design an algorithm for insertion algorithm and obtain its time complexity. Apply insertion sort on these elements. 89, 45, 68, 90, 29, 34, 17	10	L3	CO2

3.a

- **Divide-and-conquer strategy splits the inputs into k distinct subsets,1< k < n, yielding k sub-problems.**
- **These sub-problems must be solved,**
- **Then a method must be found to combine sub-solutions into a solution of the whole.**
- **If the sub-problems are still relatively large, then the divide-and-conquer strategy can be reapplied.**
- **The Sub-problems are of the same type as the original problem.**
- **A recursive algorithm is used to solve the problem.**

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
    }
}
```

Merge Sort:

```
ALGORITHM Mergesort(A[0..n − 1])
    //Sorts 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])
        Merge(B, C, A)    //see below
```

ALGORITHM *Merge*($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)

//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 elements of B and C

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leq C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
  $k \leftarrow k + 1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

3.b Insertion Sort

Insertion Sort Steps:

We start from the second element (index 1) and compare it backwards.

Step 1:

Compare 45 with 89 → 45 < 89 → Swap
→ 45 89 68 90 29 34 17

Step 2:

Compare 68 with 89 → 68 < 89 → Swap
Then compare 68 with 45 → 68 > 45 → Stop
→ 45 68 89 90 29 34 17

Step 3:

Compare 90 with 89 → 90 > 89 → No change
→ 45 68 89 90 29 34 17

Step 4:

Compare 29 with 90 → Swap
Compare 29 with 89 → Swap
Compare 29 with 68 → Swap

Compare 29 with 45 → Swap
→ 29 45 68 89 90 34 17

Step 5:

Compare 34 with 90 → Swap
34 with 89 → Swap
34 with 68 → Swap
34 with 45 → Swap
34 with 29 → 34 > 29 → Stop
→ 29 34 45 68 89 90 17

Step 6:

Compare 17 with 90 → Swap
17 with 89 → Swap
17 with 68 → Swap
17 with 45 → Swap
17 with 34 → Swap
17 with 29 → Swap
→ 17 29 34 45 68 89 90

Algorithm for Insertion sort

```
for i from 1 to length(array) - 1:
    key = array[i]
    j = i - 1
    while j >= 0 and array[j] > key:
        array[j + 1] = array[j]
        j = j - 1
    array[j + 1] = key
```

Q.4	a.	Design an algorithm for Quick sort. Apply quick sort on these elements. 5, 3, 1, 9, 8, 2, 4, 7.	10	L3	CO2
	b.	Explain Strassen's Matrix multiplication and derive its time complexity.	10	L2	CO2

4.a.

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 and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow Partition(A[l..r])$ //s is a split position
$Quicksort(A[l..s-1])$
$Quicksort(A[s+1..r])$

ALGORITHM *HoarePartition*($A[l..r]$)
//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r ($l < r$)
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r+1$
repeat
repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
swap($A[i], A[j]$)
until $i \geq j$
swap($A[i], A[j]$) //undo last swap when $i \geq j$
swap($A[l], A[j]$)
return j

Applying Quick Sort to the list: 5, 3, 1, 9, 8, 2, 4, 7

Step	Array State	Explanation
Initial	**5, 3, 1, 9, 8, 2, 4, 7**	**Choose pivot = 7 (last element)**
Partition	**5, 3, 1, 2, 4, 7, 9, 8**	**Elements < 7 to left, > 7 to right**
Left side	**5, 3, 1, 2, 4**	**Apply quick sort recursively**
Right side	**9, 8**	**Apply quick sort recursively**

Sorting left side 5, 3, 1, 2, 4 (pivot = 4)

Step	Array State	Explanation
Partition	3, 1, 2, 4, 5	Elements < 4 to left, > 4 to right
Left side	3, 1, 2	Apply quick sort recursively
Right side	5	One element, already sorted

Sorting 3, 1, 2 (pivot = 2)

Step	Array State	Explanation
Partition	1, 2, 3	Elements < 2 to left, > 2 to right
Left side	1	One element, already sorted
Right side	3	One element, already sorted

Sorting right side 9, 8 (pivot = 8)

Step	Array State	Explanation
Partition	8, 9	Elements < 8 to left, > 8 to right
Left side	Empty	No elements
Right side	9	One element, already sorted

Final Sorted Array:

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

4.b.

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

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j) \qquad (3.10)$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \qquad (3.11)$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \qquad (3.12)$$

We have 8 products and 4 sums.

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \qquad (3.13)$$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned} \qquad (3.14)$$

The resulting recurrence relation for $T(n)$ is

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

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\le cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

Module – 3

Q.5	a.	Define AVL trees. Explain its four rotation types.	10	L2	CO3
	b.	Design an algorithm for Heap sort. Construct bottom – up heap for the list 15, 19, 10, 7, 17, 16.	10	L3	CO4

5 CO3 L2

5(a) Define AVL tree. Construct an AVL tree of the list of keys: 3, 6, 5, 1, 2, 4 indicating each step of key insertion and rotation.

An **AVL tree i**s a **self-balancing binary search tree (BST)** where the **difference in height** between the left and right subtrees (called the **balance factor**) of **any node is at most 1**.

Key Properties:

- For every node in the tree:
Balance Factor = Height of Left Subtree - Height of Right Subtree
and it must be -1, 0, or +1.

- Whenever an insertion or deletion operation causes the balance factor to go outside this range, the tree performs **rotations** (single or double) to restore balance.

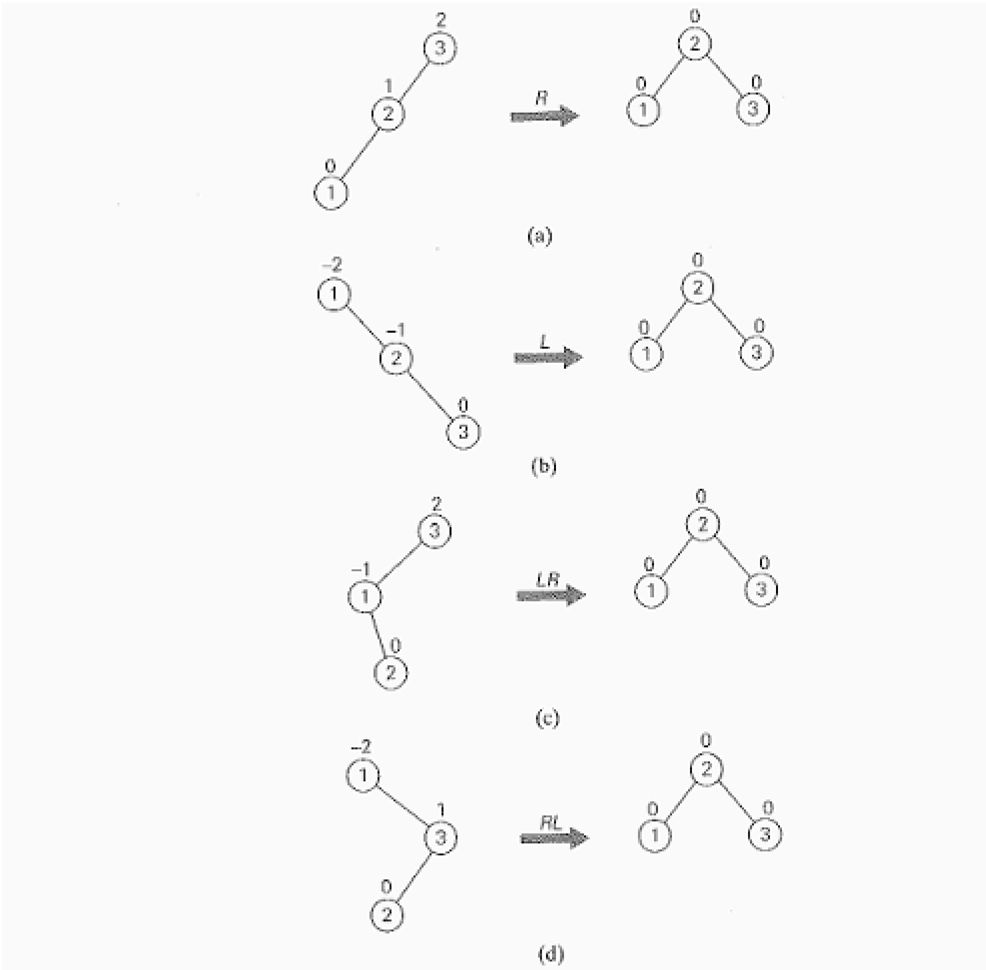


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

```
5B,HEAPSORT(arr)
   BUILD_MAX_HEAP(arr)
   for i from length(arr) - 1 down to 1
      swap arr[0] and arr[i]
      heap_size = heap_size - 1
      MAX_HEAPIFY(arr, 0, heap_size)
```

```
BUILD_MAX_HEAP(arr)
  heap_size = length(arr)
  for i from floor(length(arr)/2) down to 0
    MAX_HEAPIFY(arr, i, heap_size)

MAX_HEAPIFY(arr, i, heap_size)
  left = 2*i + 1
  right = 2*i + 2
  largest = i
  if left < heap_size and arr[left] > arr[largest]
    largest = left
  if right < heap_size and arr[right] > arr[largest]
    largest = right
  if largest != i
    swap arr[i], arr[largest]
    MAX_HEAPIFY(arr, largest, heap_size)
```

`5, 19, 10, 7, 17, 16`

Bottom-Up Heap Construction (Heapify)

This method builds the heap by calling `heapify` from the **last non-leaf node** up to the root.

Step 1: Identify last non-leaf node

- For an array of length `n`, last non-leaf node index = `floor(n/2) - 1`
- Here, `n = 6`

- Last non-leaf index = `floor(6/2) - 1 = 2`

Step 2: Apply max-heapify from index 2 down to 0

Initial array (index):
`[15(0), 19(1), 10(2), 7(3), 17(4), 16(5)]`

Heapify process:

---

i = 2 (value = 10)

- Left child index = 2*2 + 1 = 5 → value = 16
- Right child index = 2*2 + 2 = 6 → no child (out of range)

Compare 10 with 16:
Since 16 > 10, swap:

New array:
`[15, 19, 16, 7, 17, 10]`

Heapify at index 5 (value = 10): no children → stop.

---

i = 1 (value = 19)

- Left child index = 3 → 7
- Right child index = 4 → 17

19 is already greater than both children → no change.

---

i = 0 (value = 15)

- Left child index = 1 → 19
- Right child index = 2 → 16

19 is largest among 15, 19, 16 → swap 15 and 19:

New array:
`[19, 15, 16, 7, 17, 10]`

Heapify at index 1 (value = 15):

- Left child = 3 → 7
- Right child = 4 → 17

17 > 15 → swap:

New array:
`[19, 17, 16, 7, 15, 10]`

Heapify at index 4 (value = 15): no children → stop.

Final max-heap array:

`[19, 17, 16, 7, 15, 10]`

Visual representation of the heap:

```
        19
       /    \
     17       16
    /  \       /
   7    15  10
```

6.a.

Q.6	a.	Design Horspool's Algorithm for string matching Apply Horspool algorithm to find pattern BARBER in the test: JIM_SAW_ME_IN_A_BARBERSHOP.	10	L3	CO4
	b.	Define heap. Explain the properties of heap along with its representation.	10	L2	CO3

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then

stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

ALGORITHM *ShiftTable*($P[0..m-1]$)

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//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 by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R        ■
```

6.b.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//          or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    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
```

—-----------------------------------

A **heap** is a **complete binary tree** that satisfies the **heap property**.

Types of Heap:

Max-Heap: In a max-heap, the **value of each parent node is greater than or equal to the values of its children**. -The largest element is at the root.

Min-Heap: In a min-heap, the **value of each parent node is less than or equal to the values of its children**. The smallest element is at the root.

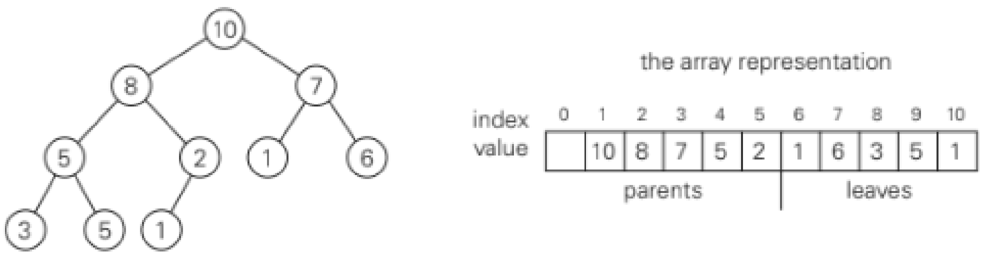


FIGURE 6.10 Heap and its array representation.

Properties of Heap:

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
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 through 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 $\lceil n/2 \rceil$ 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 $\lfloor i/2 \rfloor$.

		5	CO3	L3
7		5	CO3	L2

Module – 4

Q.7	a.	Construct minimum cost spanning tree using Kruskal's algorithm for the following graph.	10	L3	CO4

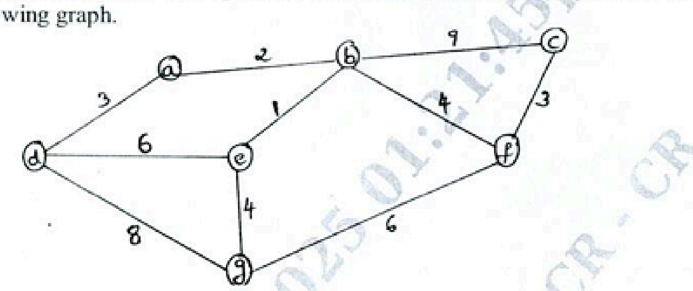


Fig. 7(a)

	b.	What are Huffman trees? Construct the Huffman tree for the following data	10	L3	CO4

Character	A	B	C	D	-
Probability	0.4	0.1	0.2	0.15	0.15

i) Encode the text ABAC ABAD

ii) Decode the code100010111001010

7.a. Kruskal's Algorithm

7.b.

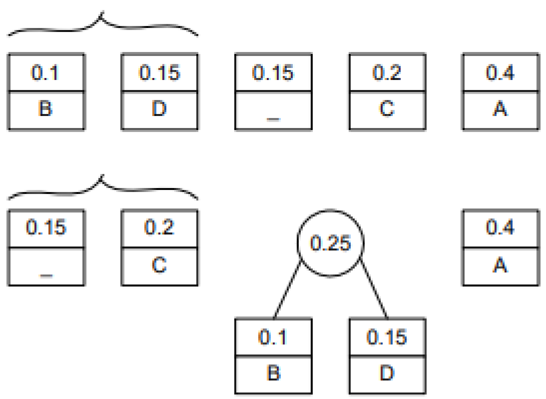
Tree edges	Sorted list of edges	Illustration/Reason
{}	bc ab ac cd be ae de ad ce	Start with empty tree
bc	bc ab ac cd be ae de ad ce	Add bc (weight 1)
bc, ab	ab ac cd be ae de ad ce	Add ab (weight 2)
bc, ab, ac	ac cd be ae de ad ce	Add ac (weight 3)
bc, ab, ac, cd	cd be ae de ad ce	Add cd (weight 3)
bc, ab, ac, cd, be	be ae de ad ce	Add be (weight 4)
bc, ab, ac, cd, be, ae	ae de ad ce	Add ae (weight 4)
bc, ab, ac, cd, be, ae	de ad ce	Skip de (6), ad (8), ce (9) - form cycles

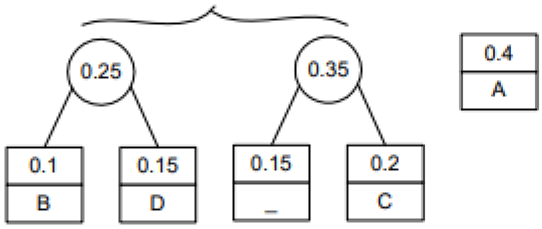
Final MST edges:

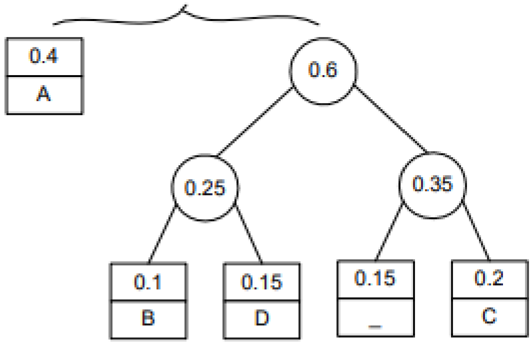
bc, ab, ac, cd, be, ae

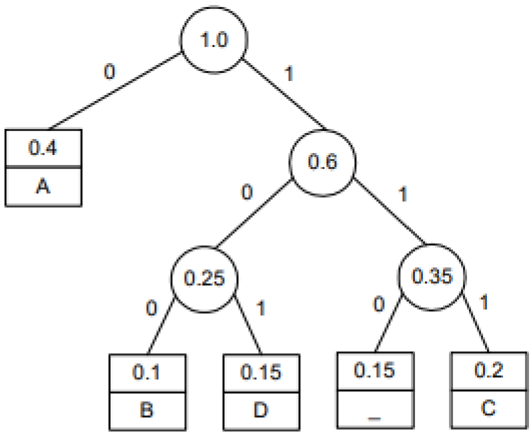
Total cost:

1 + 2 + 3 + 3 + 4 + 4 = 17









character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15
codeword	0	100	111	101	110

b. The text `ABACABAD` will be encoded as 0100011101000101.

OR

Q.8	a.	Apply Dijkstra's algorithm to fine single source shortest path for the given graph by considering A as the source vertex.	10	L3	CO4

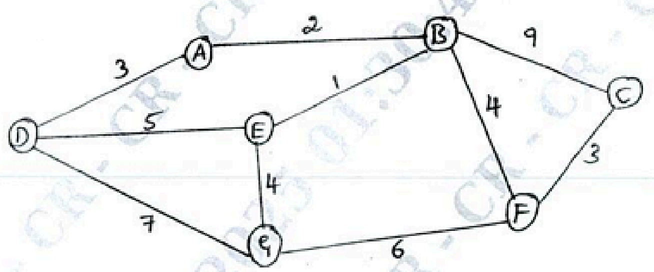


Fig.8 (a)

	b.	Define transitive closure of a graph. Apply Warshall's algorithm to compute transitive closure of a directed graph.	10	L3	CO4

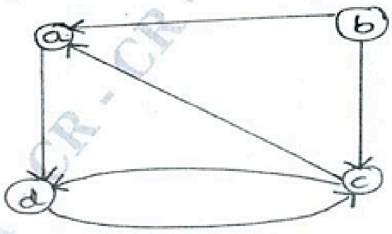


Fig.8 (b)

Step-by-step walkthrough:

Step	Current Node	Distances So Far (tentative)	Explanation
Init	—	A:0, B:∞, C:∞, D:∞, E:∞, F:∞, G:∞	Start at A (distance 0), rest infinity
1	A	A:0, B:2, C:∞, D:3, E:∞, F:∞, G:∞	Update neighbors B (2), D (3)
2	B	A:0, B:2, C:11, D:3, E:3, F:6, G:∞	Update neighbors: C=2+9=11, E=2+1=3, F=2+4=6
3	D	A:0, B:2, C:11, D:3, E:3, F:6, G:10	Update neighbors: E=3 (already 3), G=3+7=10
4	E	A:0, B:2, C:11, D:3, E:3, F:6, G:7	Update G: min(10, 3+4=7) = 7

5	G	A:0, B:2, C:11, D:3, E:3, F:6, G:7	Check neighbors, no better paths
6	F	A:0, B:2, C:9, D:3, E:3, F:6, G:7	Update C: min(11, 6+3=9) = 9
7	C	A:0, B:2, C:9, D:3, E:3, F:6, G:7	No further updates

Final shortest distances from A:

Node	Distance
A	0
B	2
C	9
D	3
E	3
F	6
G	7

Transitive Closure Definition:

The **transitive closure** of a relation **R** on a set is the **smallest transitive relation** that contains **R**.

For a directed graph **G = (V, E)**, the **transitive closure** of **G** is a graph G^+ **= (V,** E^+**)** such that for every pair of vertices **(u, v)**:

- **(u, v) ∈** E^+ if and only if **there is a path from u to v** in **G**.

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

Module – 5

Q.9	**a.**	Explain the following with examples. i) P problem ii) NP problem ii) NP-Complete problem iv) NP – Hard problem	**10**	L2	**CO5**
	b.	What is backtracking? Apply backtracking to solve the below instance of sum of subset problem. S = { 1, 2, 5, 6, 8} and d = 9.	**10**	L3	**CO6**

9.a.

i) P Problem (Polynomial time problem)

Definition: Problems that can be solved by an algorithm in polynomial time, i.e., the time taken to solve the problem grows polynomially with the size of the input.
Example: Sorting a list of numbers using Merge Sort or Quick Sort (time complexity O(n log n)).

---

ii) NP Problem (Nondeterministic Polynomial time problem)

Definition: Problems for which a proposed solution can be verified in polynomial time, but it is not necessarily known if they can be solved in polynomial time.
Example: The **Subset Sum Problem**: Given a set of integers, is there a subset that sums to zero? Verifying a subset is quick, but finding it might be hard.

---

iii) NP-Complete Problem

Definition: Problems that are both in NP and as hard as any problem in NP. If any NP-Complete problem can be solved in polynomial time, then all NP problems can be. They are the hardest problems in NP.
Example: The **Traveling Salesman Problem (TSP)**: Finding the shortest possible route that visits each city exactly once and returns to the origin city.

---

iv) NP-Hard Problem

Definition: Problems that are at least as hard as NP-Complete problems but are not necessarily in NP themselves. They might not have solutions verifiable in polynomial time.

Example: The **Halting Problem**: Determining whether a program halts or runs forever is undecidable, and thus NP-Hard.

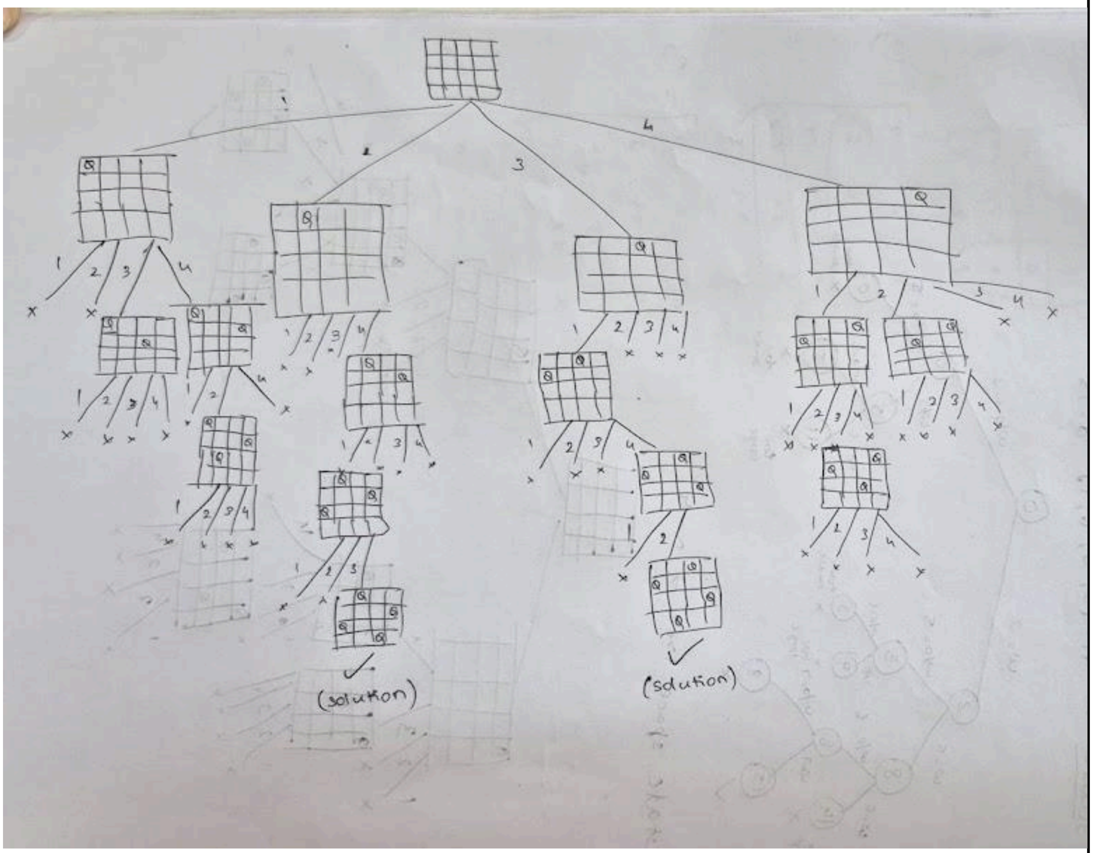
Q.10	a.	Illustrate N Queen's problem using backtracking to solve 4 – Queens problem.	10	L2	CO6
	b.	Using Branch and Bound method solve the below instance of Knapsack Problem.	10	L3	CO6

Item	Weight	Value
1	4	40
2	7	42
3	5	25
4	3	12

Capacity = 10

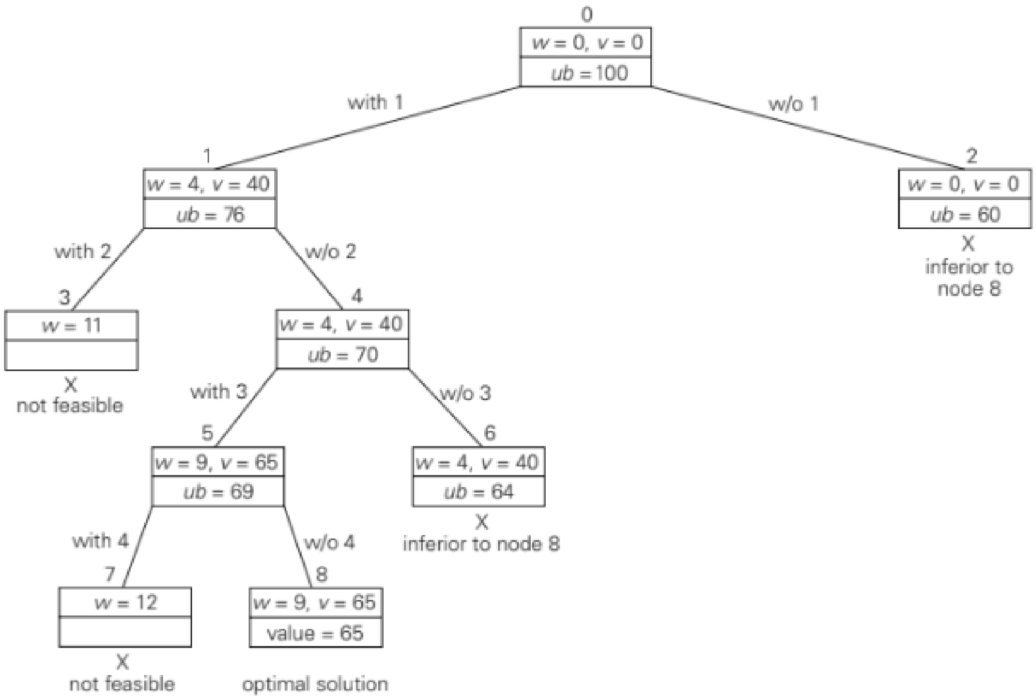
* * * * *

10.a.



10.b.

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4



CI **CCI** **HOD**