CBCS SCHEME

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

BCS401

Fourth Semester B.E./B.Tech. Degree Examination, June/July 2025
Analysis and Design of Algorithms

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		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
		OR			
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
		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
		OR			
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
		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
		OR			
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

Module – 4

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

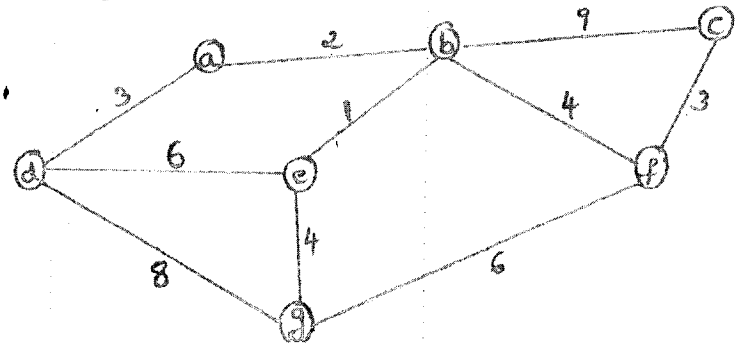


Fig. 7(a)

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

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**

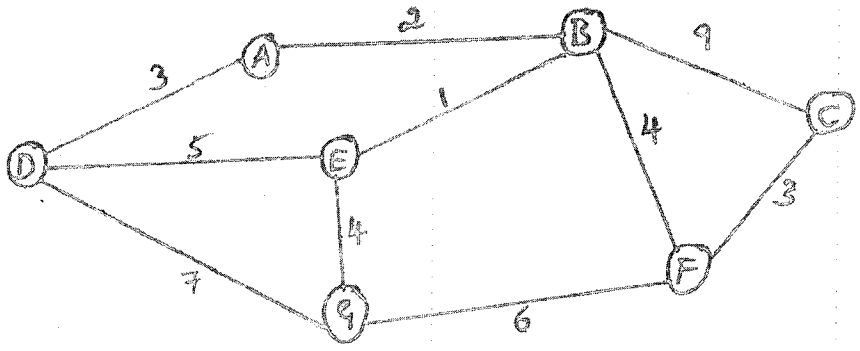


Fig.8 (a)

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

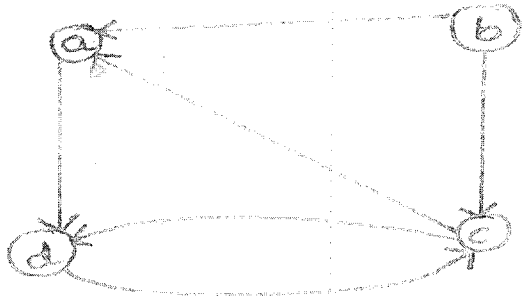


Fig.8 (b)

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
		OR			
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. Capacity = 10	10	L3	CO6

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

* * * * *

Analysis and Design of Algorithm

BCS401

Scheme of Solution

1a. 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.

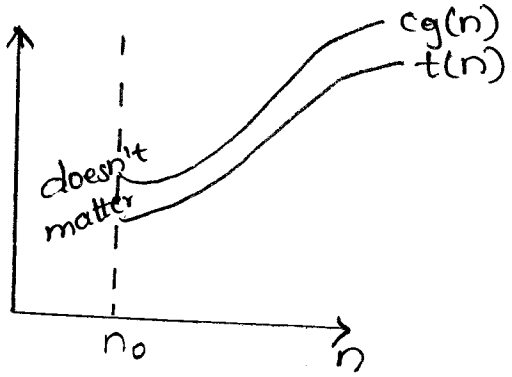
Asymptotic Notations:

To compare and rank such order of growth, computer scientists use three notations : 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 by $t(n) \in O(g(n))$, if $t(n)$ is bounded above bysome constant multiple of $g(n)$ for all large n i.e if there exist some +ve constant c and same non-ve integer n_0 such that

$$t(n) \leq c\,g(n) \quad \text{for all } n \geq n_0$$



$t(n) \in O(g(n))$

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

$$100n + 5 \leq 100n + n \quad \forall\ (n \geq 5) = 101n \leq 101n^2$$

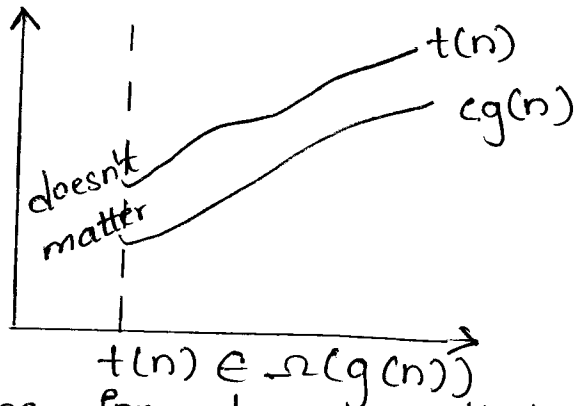
Thus an values of constant c & n_0 required by the def^n we can take 101 & 5 respectively. Note that the def^n gives us a lot of freedom in choosing specific values of constant c & n_0

$100n + 5 \leq 100n + 5n \quad (n \geq 1) = 105n \quad c = 105$ and $n_0 = 1$

Ω notation:

Definition: A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$ is bounded below by some +ve constant multiple of $g(n)$ for all large n i.e if there exist some +ve constant c and same non-ve integer n_0 such that

$$t(n) \geq c\,g(n) \quad \forall \; n \geq n_0$$



Ex. of the formal proof that $n^3 \in \Omega(n^2)$

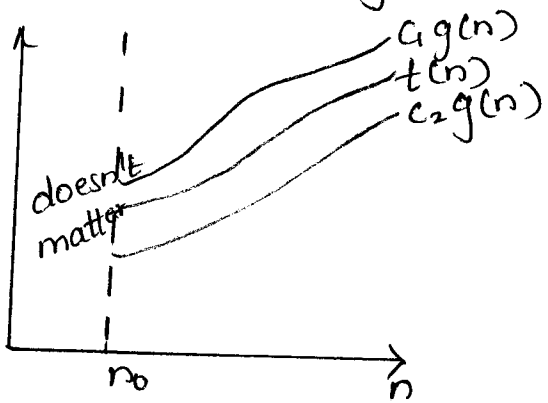
$$n^3 \geq n^2 \quad \forall \; n \geq 0$$

i.e we can select $c=1$ & $n_0=0$

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 by both above & below by same +ve constant multiple of $g(n)$ for all integer large i.e if there exist same +ve constants c_1 & c_2 and some non-ve integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \forall \; n \geq n_0$$



Example, Let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$, 1st we prove that right inequality (the upper bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \forall\ n \geq 0$$

2nd, we prove the left inequality (lower bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 = \frac{1}{2}n\,\frac{1}{2}n \quad (\forall\ n \geq 2)$$

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

Hence we select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$ and $n_0 = 2$

1. b) General plan for analyzing efficiency of recursive Algorithm

1. Decide on parameter n indicating input size.
2. Identify algorithm basic operation.
3. Check whether the no. of times the basic operation is executed depends only on the input size n. It is depends on the type of input, investigate worst, average and best case efficiency seperately.
4. Set up recurrence relation, with an appropriate initial cond^n for the no. of times the algorithms basic operation is executed.
5. Solve the recurrence.

Factorial Function

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

Recursive definition of n!:

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

$F(0) = 1$

Algorithm Factorial(n)

// purpose - Compute n! factorial resursively

// Input - A non-ve integer n

// output - The value of n!

```
if (n=0)
    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 no. 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.

5. Solve the recurrence relation using backward substitution method.

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

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

$= M(n-2) + 2$

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

$= M(n-3) + 3.$

The general formula for above pattern for same 'i' as follow

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

By taking the advantage of initial cond^n given i.e $M(0) = 0$. now $n - i = 0$ and obtain $i = n$.

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

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

$$= M(0) + n$$

$$M(n) = n.$$

$\therefore M(n) \in \Theta(n).$

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

1. c) Given: $t_1(n) \in O(g_1(n))$

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

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

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

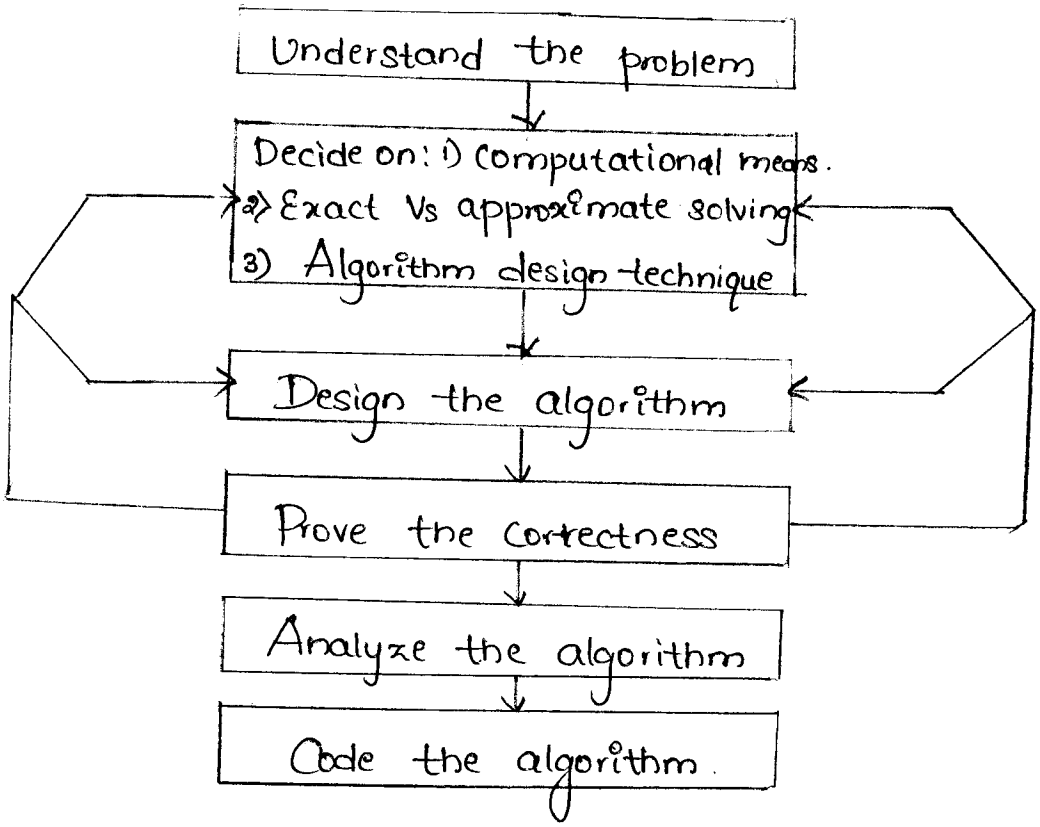
$t_1(n)+t_2(n) \leq c_1(g_1(n))+c_2(g_2(n))$

$\leq 2c_3 \max(g_1(n), g_2(n))$

$\leq \max\{2c_3(g_1(n), 2c_3(g_2(n))\}$

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

2a.



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 instance the algo. needs to handle else, it will work correctly for majority of input but crash on some boundary value.

* A correct algorithm in not done that works most of the time, but one that works correctly for all legitimate input.

Ascertaining of Capabilities of a Computational device.

* After a 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 Neuman machine a computer architecture.
* Von Neunmann machines are sequential & the algorithms implemented on them are called Sequential algorithms.
* Algorithms designed to be ~~extend~~ executed on parallel Computer are called parallel algorithms.
* Complex algorithms concentrate on a machine with high speed & more memory where time is critical.

Choosing between exact and approximate problem solving.

* for exact result - exact algorithm.
* For approximate result - approximat^n algorithm.
* Example of exact algorithms obtaining Square root for number & Solving non-linear equat^n.
* An approximation algorithm can be part of a more sophisticated algorithm that solves a problem exactly.

Deciding an appropriated data structures.

* Algorithms may or may not demand ingenuity in representing the inputs.
* Inputs are represented using various data structure.
* 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 i.e applicable to a

variety of problems from different areas of computing.

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

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 reqd result for every legitimate input in a finite amount of time.
* The correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality $gcd(m,n) = gcd(n, m \bmod n)$, the simple observatn that second integer gets smaller on every interaction of the algorithm & the fact that the algorithm stops when the second integer becomes 0.
* Some algorithm, a roof of correctness is quite easy for others it can be quite complex.
* 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 has to be estimated.
* Time efficiency & Space efficiency.
* Time: how fast the algorithm runs?
* Space: how much extra memory the algorithm needs?
* Simplicity: how simpler it is compared to existing algorithm?
* Generality: Generality of the problem the algorithm solves & the set of inputs it accepts.
* If not satisfied with the 3 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 verificatn. But while implementing algo. as programs to be used in actual applⁿ, it is required to provide such verification.

2b. Finding the max. element in an array of n elements.

```
Algorithm maxElement (A[0...n-1])
// Determines the value of the largest element in an given array.
// Input : An Array A[0..n-1] of real no.
// 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 base operation i.e
   * Comparison : A[i] > maxval
   * Assignment : maxval ← A[i]

   Here the comparison statement is considered to be the basic operatn of the algorithm.
3. No best, worst, average case because the no. of comparisons will be same for all arrays of size n & it is not dependent on type of input.
4. Let C(n) denotes no. of comparison : Algorithm makes one comparison on each execut of the loop which is repeated for each value of the loop's variable i within the bound b/n 1 & 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.

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

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

$$\left[\sum_{i=1}^{u} 1 = u-l+1 \right]$$

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

2. C. Algorithm selection sort (A[0...n-1])

// sorts a given array by selection sort.

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

// output: Array A[0...n-1] sorted in increasing order.

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

Analysis:

Input: Input size is given by the no. of elements n.

Basic operation: key comparison A[j] < A[min]

Basic operation count: The no. of times the basic operation is executed depends only on the array size & given by following sum.

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

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

$$= \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$$

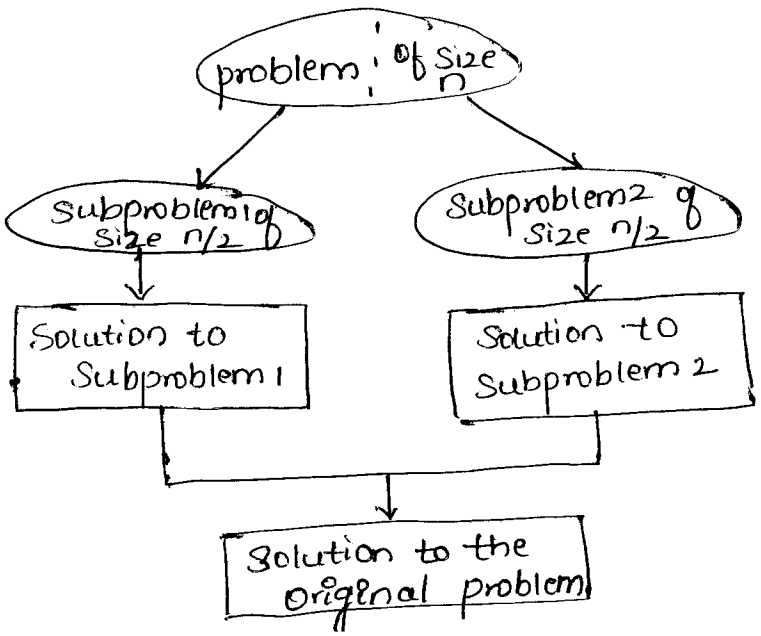
$$= \frac{(n-1)n}{2} = n^2 - n = n^2$$

Thus selection sort is $O(n^2)$ algorithm on all inputs.

3a. Divide and conquer is probably the best known generatn algorithm design technique. The algorithm works according to following general plan.

1. A problem is divided into several subproblems of same type ideally about equal size
2. The subproblem are solved.
3. If necessary, the solution to the subproblems are combined to get a soln to original problems.

The divide and conquer technique depicts the case of dividing a problem into 2 smaller subproblems, the most widely occuring case.



Merge sort (A[0...n-1])

//Sorts array A[0...n-1] by recursive merge sort.

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

//Output: Array A[0...n-1] sorted in non decreasing 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)
```

Merge (B[0...p-1], c[0...q-1], A[0...p+q-1]

//Merges two sorted Array into one sorted array

//Input: Arrays B[0...p-1] & c[0...q-1] both sorted

//output: Sorted array A[0...p+q-1] of element of B&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                j←j+1
        A[k] ← c[j]
        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]
```

Analysis: Assuming for simplicity that n is a power of 2 the recurrence relation of the no. of key comparisons

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n>1$$

$$C[1] = 0$$

Let us analyse $C_{merge}(n)$, the no. of key comparisons performed during the merge sort.

At each step, exactly one comparison is made, after which the total no. of elements in the 2 arrays still needing to be processed is reduced by 1.

* In worst case, neither of 2 arrays becomes empty before the other one contains just one element.

∴ worst case, $C_{merge} = n-1$ & have recurrence.

$$C_{worst}(n) = 2 C_{worst}(n/2) + n - 1 \quad \text{for } n > 1$$

$$C_{worst}(1) = 0$$

Hence theorem (Master theorem)

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

Analysis of merge sort using master theorem.

If the time for merging operation is proportional to n, then

$$T(n) = \begin{cases} a & n = 1, \ a \text{ is constant} \\ 2T(n/2) + c \cdot n & n \geq 1 \quad c \text{ is constant.} \end{cases}$$

Assume $n = 2^k$ then

$$T(n) = 2 \cdot T(n/2) + c \cdot n$$

$$= 2 \cdot [2 \cdot T(n/2) + c \cdot n/2] + c \cdot n$$

$$= 4T(n/4) + 2cn.$$

$$= 8T(n/8) + cn + 2cn$$

$$= 2^3 T(n/2^3) + 3c \cdot n$$

$$= 2^k T(n/2^k) + k\,cn$$

$$= nT(1) + k\,cn. \quad \because n = 2^k$$

$$T(n) = na + cn\log n$$

if $2^k \leq n \leq 2^{k+1}$ then $T(n) \leq T(2^{k+1})$

So, $T(n) = O(n \log n)$.

3b. Algorithm Insertion sort (A[0...n-1])

// sort a given array by insertion sort.

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

//output: Array A[0...n-1] sorted in nondecreceasing 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
```

Insertion sort algorithm for first elements.

* 1st value is already in correct position. 2nd value must be Compared & moved past the 1st value, 3rd value must be compared and moved past two values and so on.

If we continue this pattern, The total no. of operations for n values

$$1+2+3+\cdots+(n-1)$$

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

For very large n, $\frac{n^2}{2}$ term dominates, simplify by removing the 2nd term $\frac{n}{2}$

O notation, time complexity $O(n^2/2) = O(\frac{1}{2} \cdot n^2) = O(n^2)$.

Insertion sort on elements 89, 45, 68, 90, 29, 34, 17

Step by step sorting

* Start with assuming the first element is already sorted.

89, | 45, 68, 90, 29, 34, 17

* Insert the next element (45) in its correct posit^n in sorted part

45, 89, | 68, 90, 29, 34, 17

* Insert the next element (68) in its correct position in the sorted part.

* 45, 68, 89, | 90, 29, 34

* Continuing this process for remaining elements.

45, 68, 89, 90, | 29, 34

29, 45, 68, 89, 90, | 34

Final sorted array ⇒ 29, 34, 45, 68, 89, 90.

4a. Quick sort is the important sorting algorithm i.e based on the divide & conquer approach.

* Quick sort divides its input demands according to their values.
* It is divided into partition i.e A partition is an arrangement of the array's element so that all the problem to left of some element A[s] are less than or equal to A[s] & 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]

* After parition, A[s] will be in the final position in the sorted array & continue sorting the 2 subarrays to the left & to right of A[s] independently.

```
QuickSort(A[l.....r])
// Sorts a subarray by quicksort
// Input : Subarray of array A[0---n-1] defined by its left & right
           indices l & r.
// Output : Subarray A[l.....r] sorted in ascending order
    if l < r
        s ← partition(A[l---r]
        Quicksort(A[l---s-1])
        Quicksort(A[s+1,----r])
```

Hoare's partition (A[l---r])

// partition of subarray by Hoare's algo. using the 1st element as a pivot.

// Input: Subarray of array A[0---n-1] defined by its left & right indices l & r (l<r)

// ouput: Partition of A[l---r] with split position returned as this function's value.

```
p ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ p
    repeat j ← j-1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])
swap(A[l], A[j])
return j.
```

Ex: Quick sort 5, 3, 1, 9, 8, 2, 4, 7

0	1	2	3	4	5	6	7
Elements to sort							
5 (pivot no.)	3 (i)	1	9	8	2	4	7 (j)
5	3	1	9 (i)	8	2	4 (j)	7
Compare i & j value to p.no.							
5	3	1	4 (i)	8	2	9 (j)	7
Swap i & j value.							
5	3	1	4	8 (i)	2 (j)	9	7
5	3	1	4	2 (i)	8 (j)	9	7
5	3	1	4	2 (j)	8 (i)	9	7
Swap of pivot no. As j crosses i							
2	3	1	4	5	8	9	7

2	3 (i)	1	4 (j)	8	9 (i)	7 (j)	
2	3 (i)	1 (j)	4	8	7 (i)	9 (j)	
2	1 (j)	3 (i)	4	8	7 (j)	9 (i)	
1	2	3	4	7	8	9	
1				7			
		3	4 (i,j)			9	
			4				
1	2	3	4	5	7	8	9

4b. Strassen's Matrix Multiplication

* Multiplication of 2x2 matrices, by using divide & conquer method or approach, reduce the no. of multiplication such an algorithm was published by V Strassen in 1969.
* The principal insight of algorithm lies in the discovery that find the product of C of 2, 2x2 matrix A & B with just 7 multiplication as opposed to 8 requires by brute force algorithm.
* This is accomplished by using following formula.

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

$$= \begin{bmatrix} M_1+M_4-M_3+M_7 & M_3+M_5 \\ M_2+M_4 & M_1+M_3-M_2+M_6 \end{bmatrix}$$

where,

$M_1 = (a_{00}+a_{11}) * (b_{00}+b_{11})$

$M_2 = (a_{10}+a_{11}) * b_{00}$

$M_3 = a_{00} * (b_{01}-b_{11})$

$M_4 = a_{11} * (b_{10}-b_{00})$

$M_5 = (a_{00}+a_{01}) * b_{11}$

$M_6 = (a_{10}-a_{00}) * (b_{00}+b_{01})$

$M_7 = (a_{01}-a_{11}) * (b_{10}+b_{11})$

* multiply 2x2 matrix, Strassen's algorithm makes seen multiplicath 18 additions/ Subtractions where as brute force algorithm requires 8 multiplication & 4 addition.

Multiplication of nxn matrices.

* Let A & B be 2 nxn matrices where n is a power of 2.
* We divide A, B & their product C into 4 $n/2 * n/2$ Submatrices each as follows.

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

Analysis.

If M(n) is the no. of multiplication made by strassen's algo. in multiplying 2 nxn matrices, following recurrence relation for it

$$M(n) = 7M(n/2) \quad \text{for } n>1,\ M(1)=1$$

Since $n = 2^k$

$$M(2^k) = 7M(2^{k-1})$$
$$= 7[M(2^{k-2})]$$
$$= 7^2 M(2^{k-2}) \ldots$$
$$= 7^k M(2^{k-k})$$
$$= 7^k M(2^{k-k})$$
$$= 7^k$$

Since $k = \log_2 n$

$$M(n) = 7^{\log_2 n}$$
$$= n^{\log_2 7}$$
$$M(n) \simeq n^{2.807}$$

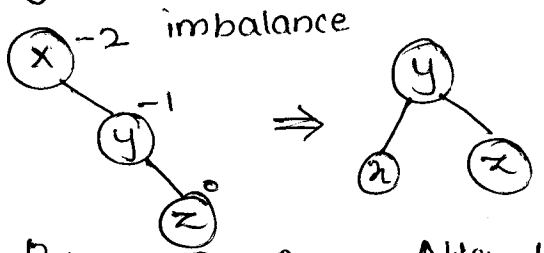
$M(n) \simeq \Theta(n^{2.807})$ is smaller than n^3 reqd by brute force algorithm.

5a. An AVL tree is an ordered binary search tree in which the height of the 2 subtrees of every node differ max. by 1 i.e the height of the left subtrees, the height of the right Subtree can be 0, 1 or -1.

Types of rotations.

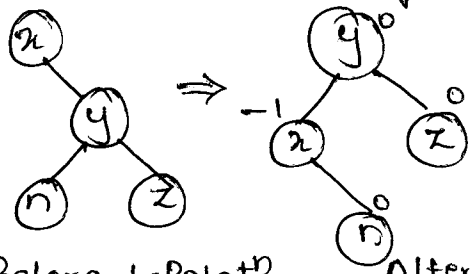
1. L Rotation: If the 3 nodes identified by step1 & step2 are in a straight line, single rotation is required. If the balanced factor of a node, where imbalance occurs is 2, then the tree is heavy, towards right so to balance it we rotate left called left rotation

Case1:



Before L-Rotatn After L-rotation

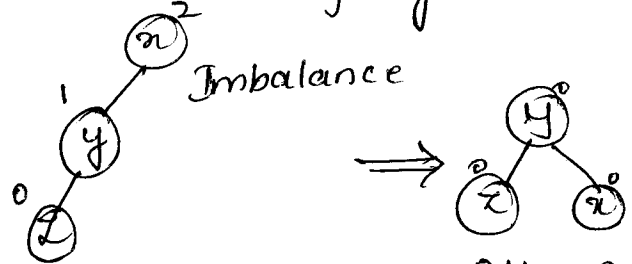
Case 2: This is left Child for y.



Before L-Rotatn After L-rotation.

2. R-Rotation ; If the 3 nodes identified in step1 & step2 are in a straight line, single rotatn is required. If the balance factor of a node where imbalance occur is 2, then tree is heavy towards left. So to balance it rotate right called right rotation.

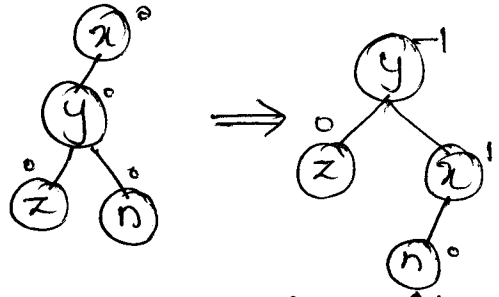
Case1 : No child for y



Before right rotatn After right rotation.

Case 2: This is right child for y

case 2: This is right child for y.

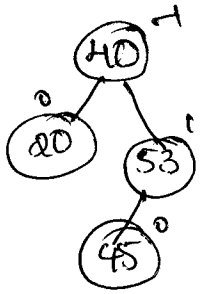


Before right rotatn After right rotatn.

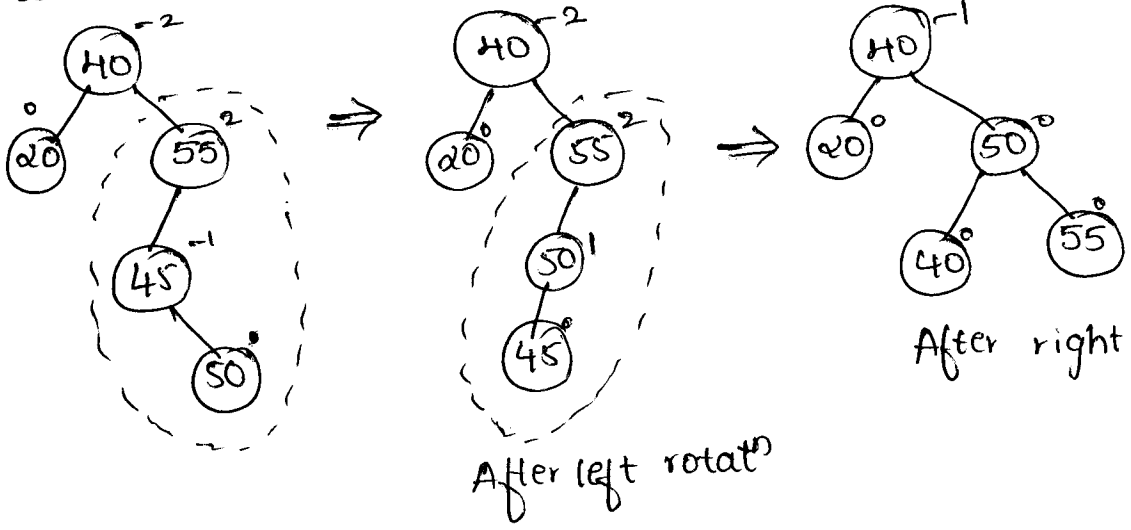
3. L-R rotation: If the 3 nodes identified are not in a Straight line, then double rotatn is reqd. A double rotatn is a combinatn of 2 Single rotatn.

1. Assume 'x' is a node where imbalance occurs & y is its child in the path. If the balance factor if y is -1, this subtree is right heavy. So rotate left at y. Attach the parent of resulting Subtree to x.

Ex:



Now insert to



After left rotatn

After right rotatn.

4. R-L rotation. If the 3 nodes identified are not in a straight line then double rotatn is required.

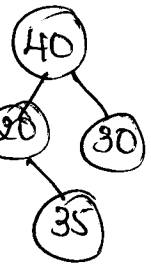
A double rotation is the combination of 2 single rotation as

i) Assume 'x' is a node where imbalance occur & 'y' is its child in path. If the balance factor of y is 1, then

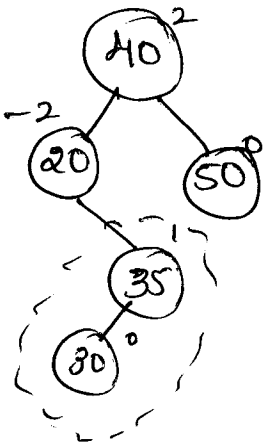
subtree is left heavy. So rotate right of y. Attach the parent of resulting subtree to x.

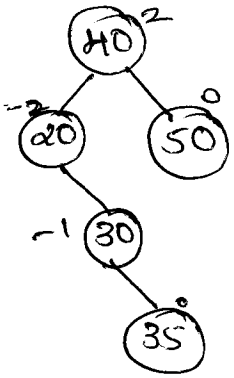
ii) 2nd rotatn is required at x. Here left rotation is required

Ex:-

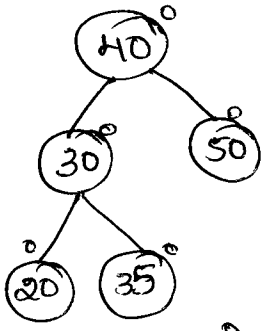


Now insert element is





R. Rotatn



L Rotation

Tree is balanced.

5b. Algorithm Heapsort(A)

```
BUILD-MAX-HEAP(A)
for i = A.length downto 2
  exchange A[1] with A[i]
  A.heapsize = A.heap-size - 1
  MAX-HEAPIFY(A,1)
```

Bottom up Algorithm:

```
Heap-Bottomup(H[1--n])
// construct a heap from the elements of a given
// array by bottom up algorithm.
// Input: An array H[1---n] for orderable items.
// output: A heap H[1---n]
    for i ← [n/2] downto 1 do
        k ← i;
        v ← H[k]
```

```
heap ← false
while not heap & 2*k ≤ n do
    j ← 2*k
    if j < n
        if H[j] < H[j+1]
            j ← j+1
        if v > H[j]
            heap ← true
        else
            H[k] ← H[j];
            k ← j
    H[k] ← v.
```

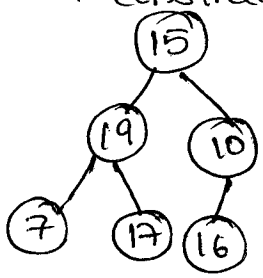
15, 19, 10, 7, 17, 16.

Bottom up construction; It initializes the essentially complete binary tree with n nodes by placing keys in the order given & the 'heapfiles' the tree as follows.

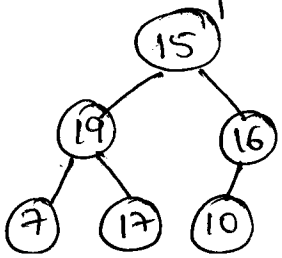
* Starting with last parental nodes, the algorithm checks whether the parental dominance holds for the key at the node.
* If it does not, the algorithm exchanges the nodes key k with the larger key of its children & checks whether the parental dominance hold for k in its new position. This process continues 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 after the node's immediate predecessor.

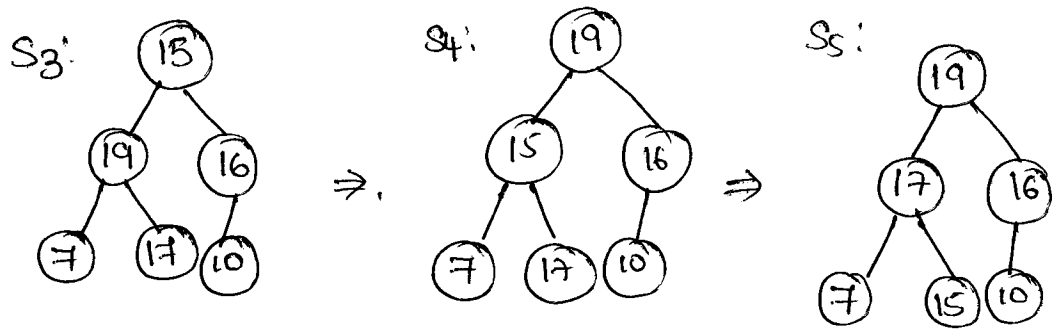
Given 15, 19, 10, 7, 17, 16

S1. Construct binary tree.



S2: Comparison Sorts.





Constructed heap: 19, 17, 16, 7, 15, 10.

6a. 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
```

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

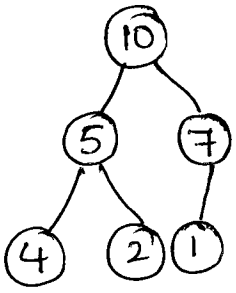
(as - v not present shift complete length) — BARBER

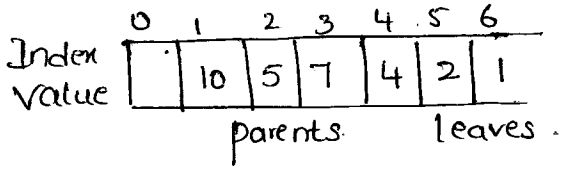
(after B 2 letter are presented left it by 2 letters). — BARBER

BARBER — Complete pattern is matched.

6b. A heap can be defined as a binary tree with keys assigned to its nodes provided the following 2 condit^{n} are met.

1. The tree's shape requirement the binary tree is essentially complete (or simply complete) i.e all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The parental dominance requirement, the key of each node is greater than or equal to the keys to its children.





array representation.

List of important properties of heap.

1. There exists exactly one esentially complete binary tree with n nodes. Its 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 convenien to store the heap's element in position 1 through n of such an array, leaving H[0] either unused to putting there a sentinel whose value is greater than every element in the heap. In such a representation

a) The parental node key will be in the first $[n/2]$ positions, of the array, while the leaf key will occupy the lost $[n/2]$ positions.

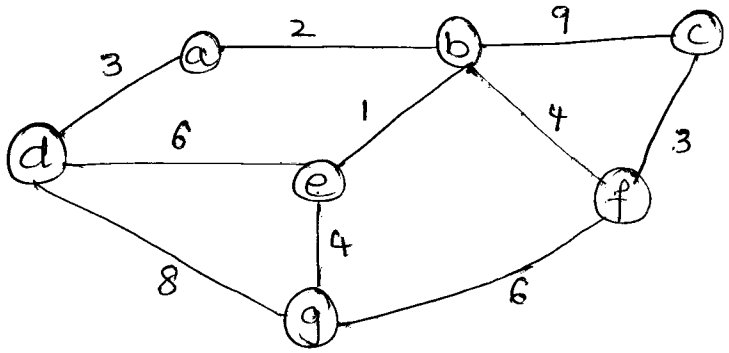
b) The children of a key in the array's parental position ($1 \le i \le [n/2]$) will be in positions $2i$ & $2i+1$ & corresponding the parent of a key in position i ($2 \le i \le n$) will be post^{n} $[i/2]$

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

7a. 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 nondecreasing order of the edge weight $w(e_{i_1}) \leq \ldots \leq w(e_{i_E})$

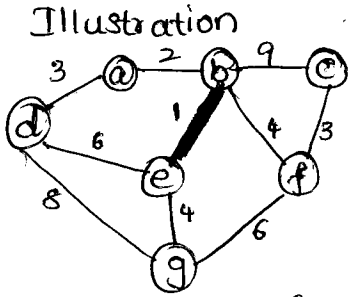
$E_T \leftarrow \phi$;
ecounter $\leftarrow 0$
$k \leftarrow 0$
while ecounter $< |V| - 1$ do
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{ik}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{Tk}\}$;
        ecounter $\leftarrow$ ecounter $+ 1$
return E_T.



Tree edges Sorted list of edges

be	ab	ad	cf	eg	bf	de	fg	dg	bc
1	2	3	3	4	4	6	6	8	9

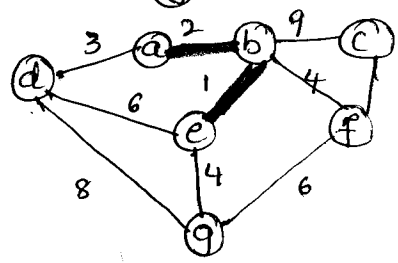
Illustration



be

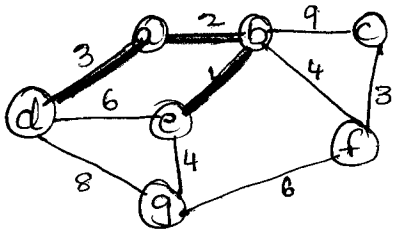
1

be	ab	ad	cf	eg	bf	de	fg	dg	bc
1	2	3	3	4	4	6	6	8	9



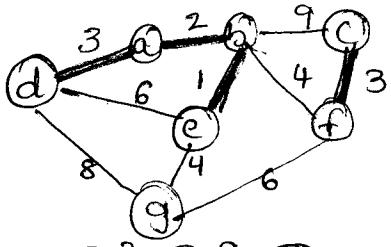
ad
3

be ab ad cf eg bf de fg dg bc
1 2 3 3 4 4 6 6 8 9



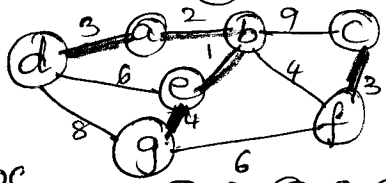
cf
3

bc ab ad cf eg bf de fg dg bc
1 2 3 3 4 4 6 6 8 9



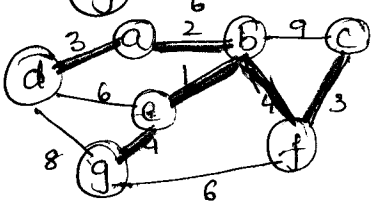
eg
3

bc ab ad cf eg bf de fg dg bc
1 2 3 3 4 4 6 6 8 9



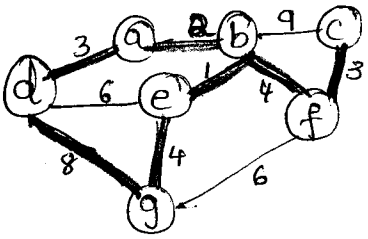
eg
4

bc ab ad cf eg bf de fg dg bc
1 2 3 3 4 4 6 6 8 9



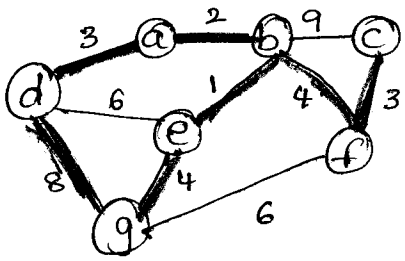
bf
4

bc ab ad cf eg bf de fg dg bc
1 2 3 3 4 4 6 6 8 9



dg
8

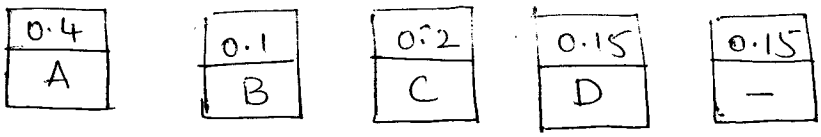
bc ab ad cf eg bf de fg dg bc
1 2 3 3 4 4 6 6 8 9



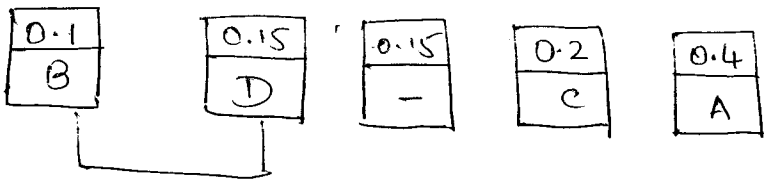
b.

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

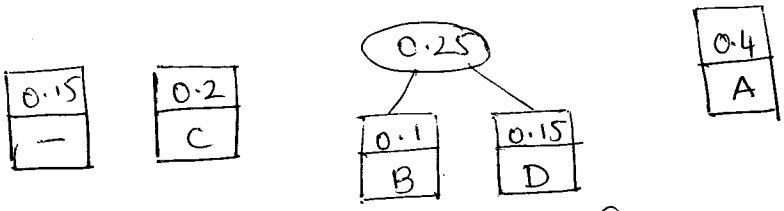
Step 1: Create N one node trees.



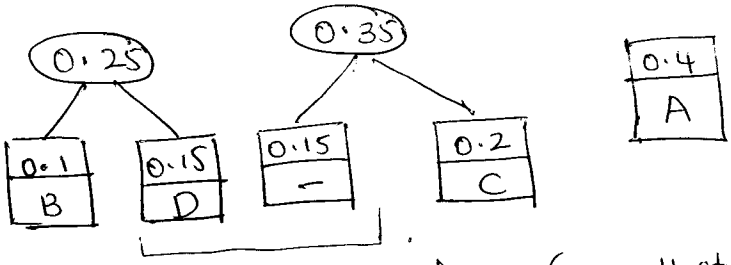
Step 2: Arrange the single node trees in ascending order



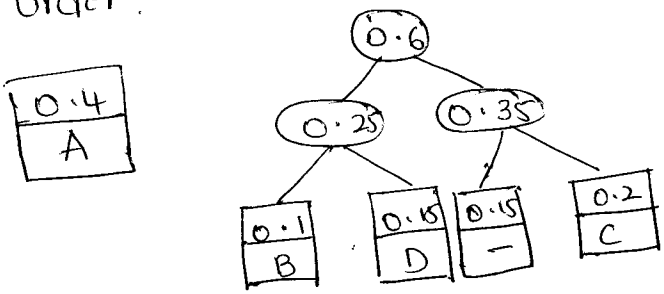
Step 3: Combine two trees (smallest value) & arrange in ascending Order.



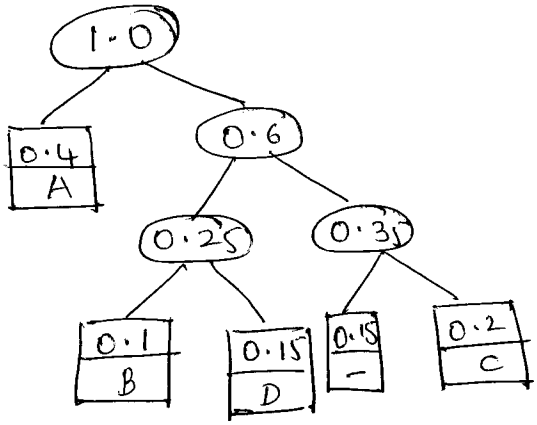
Step 4: Combine two trees (smallest value) & arrange in ascending Order.



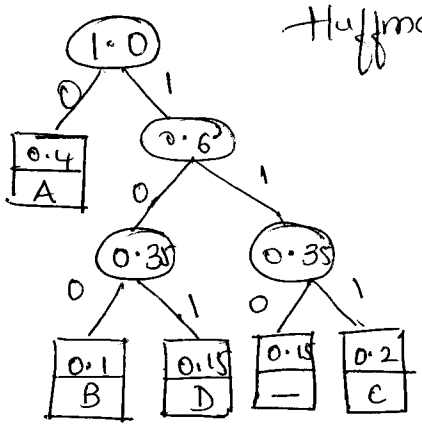
Step 5: Combine two trees (smallest value) & arrange in ascending Order.



Step 6: Combine two trees (smallest value) & arrange in ascending Order.



Character	A	B	C	D	—
Probability	0.4	0.1	0.2	0.15	0.15
codeword	0	100	111	101	110



Huffman tree

i) Encode A B A C A B A D

0 100 0 111 0 100 0 101

ii) Decode 1000 101 1100 1010

B A D — A D A

8a. Algorithm Dijkstra's ~~algorithms~~ (G, s)

// Dijsktra's algorithm for Single Source Shortest paths

// Input: A weighted connected graph $G = \{V, E\}$ with non-ve weights and its vertex s.

// output: The length d_v of a shortest path from s to v & its penultimate vertex P_v for every vertex v in V.

Initialize (Q)

for every vertex v in V

$d_v \leftarrow \infty$; $P_v \leftarrow$ null

Insert (Q, v, d_v)

$d_s \leftarrow 0$;

$V_T \leftarrow \phi$

for $i \leftarrow 0$ to $|V| - 1$ do

$u^* \leftarrow$ DeleteMin(Q)

$V_T \leftarrow V_T \cup \{u^*\}$

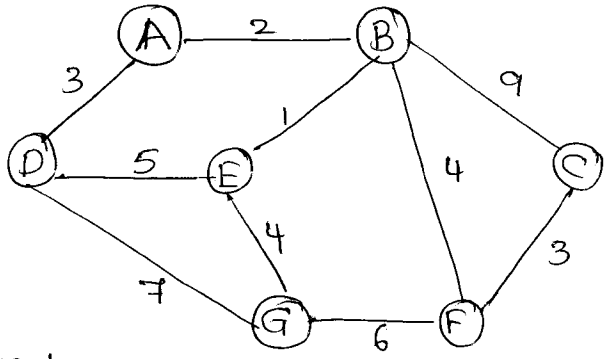
for every vertex u in $V - V_T$ i.e. adjacent to u^* do

if $d_{v^*} + w(u^*, u) < d_u$

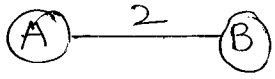
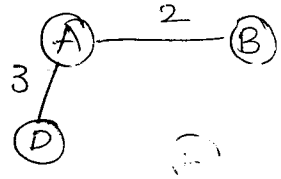
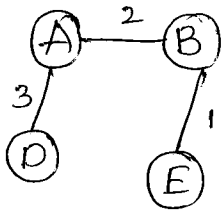
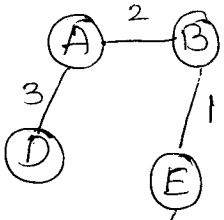
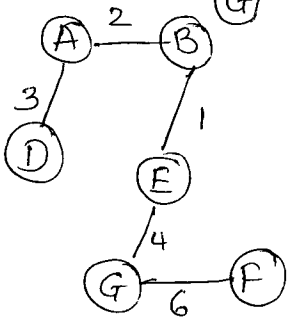
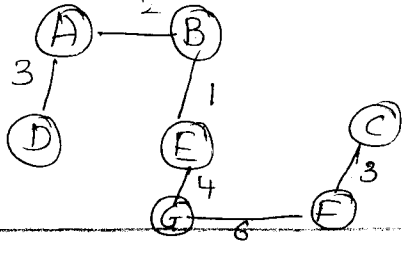
$d_v \leftarrow d_{v^*} + w(u^*, u)$;

$P_v \leftarrow u^*$

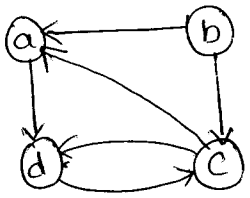
Decrease (Q, u, d_u)



Source vertex-A

Tree	Remaining Vertex	Illustratn.
A(-, ∞)	B(A,2) C(-,∞), F(-,∞) D(A,3), E(-,∞) G(-,∞)	
B(A,2)	C(B,11), E(B,3) F(-,∞) G(-,∞) ~~D(A,3)~~	
D(A,3)	G(D,10), C(B,11), ~~E(B,3)~~ F(-,∞).	
~~G~~ (B,3)	G(E,5) F(B,7)	
G(E,5)	F(G,11)	
F(B,7)	C(F,9)	

8b. Transitive Closure of a directed graph with n vertices can be defined as n×n Boolean matrix $T = \{t_{ij}\}$ in which element in the ith row & jth column is 1, if there exist a non trivial path from ith vertex to jth vertex otherwise $t_{ij} = 0$.



$R^{(0)} =$

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

Consider path through vertex a.

$b \to b = b \xrightarrow{1} a \;\&\; a \xrightarrow{0} b = 1$

$b \to d = b \xrightarrow{1} a \;\&\; a \xrightarrow{1} d = 1$

$c \to b = c \xrightarrow{1} a \;\&\; a \xrightarrow{0} b = 0$

$c \to c = c \xrightarrow{1} a \;\&\; a \xrightarrow{0} c = 0$

$d \to c = d \xrightarrow{0} a \;\&\; a \xrightarrow{0} c = 0$

$d \to d = d \xrightarrow{0} a \;\&\; a \xrightarrow{1} d = 0.$

Consider path through vertex b.

$R^{(1)} =$

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

Consider path through vertex c.

$R^{(2)} =$

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

Consider path through vertex d.

$R^{(3)} =$

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

Consider path through vertex

$R^{(4)} =$

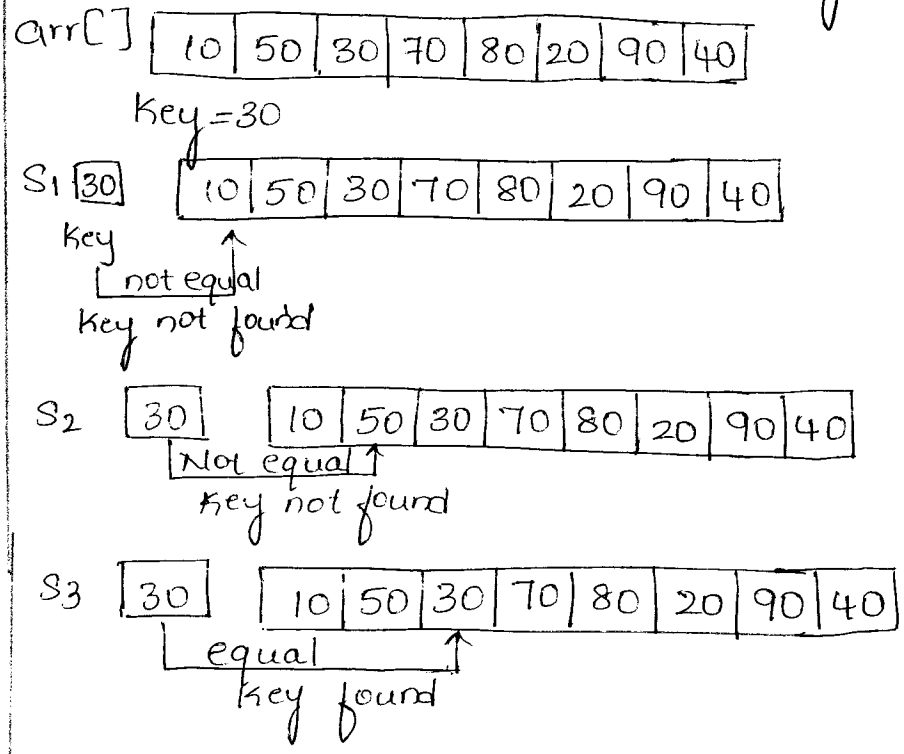
	a	b	c	d
a	0	1	1	0
b	1	0	1	1
c	1	0	1	1
d	1	0	1	1

9a.

i) P Problem: Class P is a class of decision problem that can be solved in polynomial time by algorithm this class of problem is called polynomial

Ex: Linear Search.

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



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

Example: prime factorization.

Pseudocode:

```
PrimeFactors[]
  i=0
  while n!=1;
primefactors[i]=SPF[n]
  i++
  n=n/SPF[n]
```

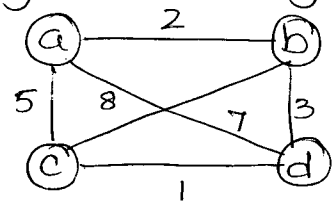
Time Complexity: O(logn), for each query (Time complexity for percomputation is not included).

iii) NP complete Problem: A decision problem D is said by to be polynomially reducible to a decision problem D_2 if there exists a funct 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 & all no instances of D_1 to no instance of D_2.
2. t is computable by a polynomial time algorithm.

Example: Travelling Salesman problem.

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

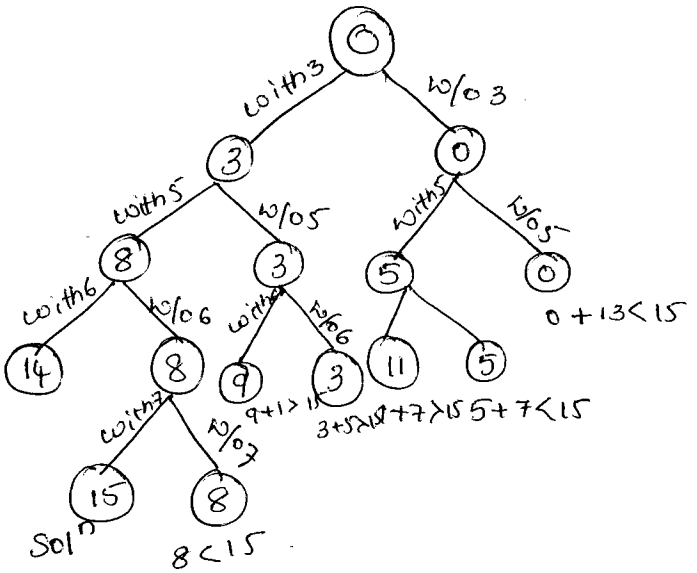


Tour	Length
$a \to b \to c \to d \to a$	$l = 2+8+1+7 = 18$
$a \to b \to d \to c \to a$	$l = 2+3+1+5 = 11$ optimal
$a \to c \to b \to d \to a$	$l = 5+8+3+7 = 23$
$a \to c \to d \to b \to a$	$l = 5+1+3+2 = 11$ optimal
$a \to d \to b \to c \to a$	$l = 7+3+8+5 = 23$
$a \to d \to c \to b \to a$	$l = 7+1+8+2 = 18$

iv) NP-Hard Problem: A 'p' pattern is said to be NP Hard when all 'Q' belonging in NP can be reduced in polynomial time (n^k where k is same constant) to p assuming a soln for 'P' takes 1 unit time.

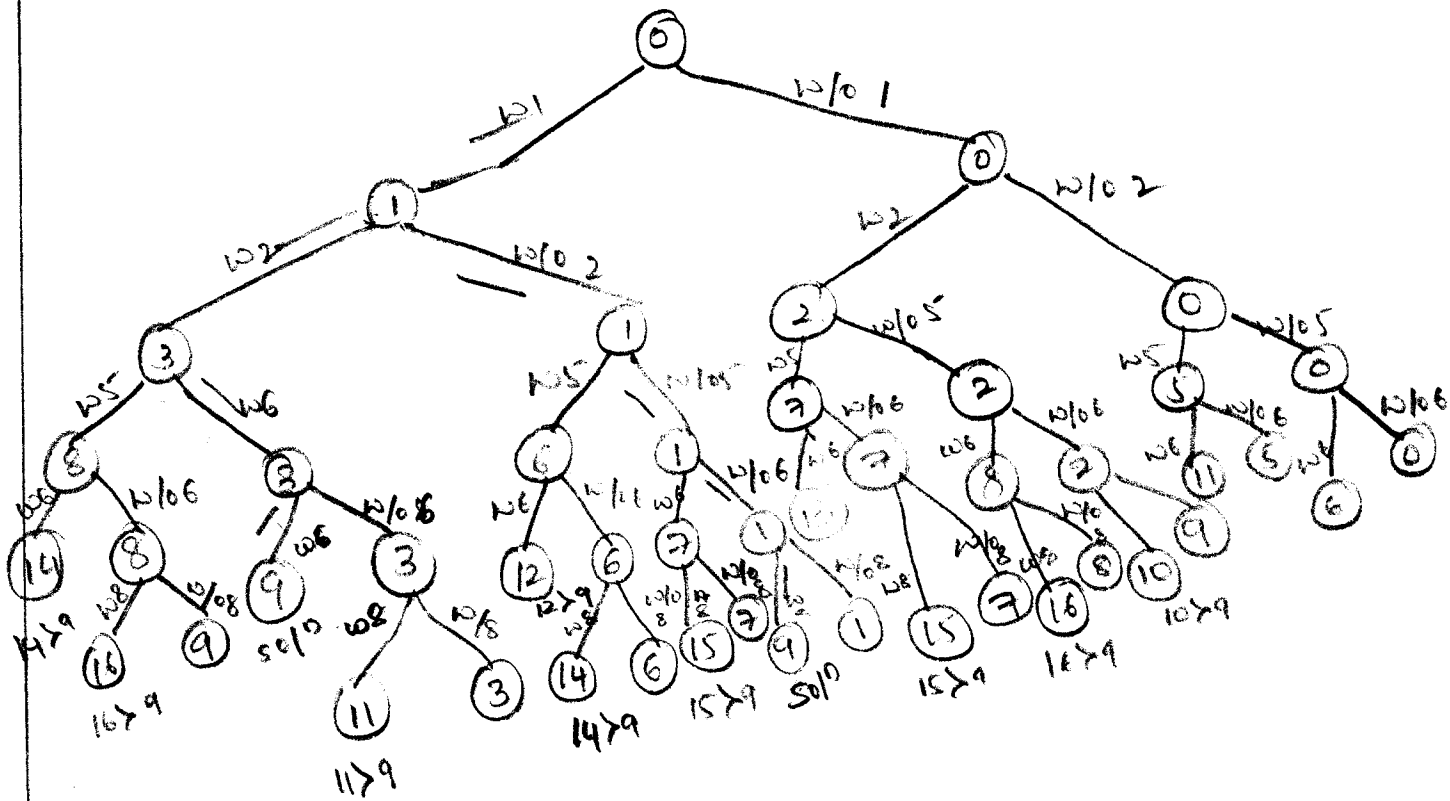
Ex: Subset Sum problem.

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



9b. Backtracking algorithms are like problem solving strateges that helps explore different options to find the best solutn. They work by trying out different paths & if one doesn't work, they backtrack & try another unit they find the right one. Its like solving a puzzle by testing different pieces until they fit together perfectly.

Given $S = \{1, 2, 5, 6, 8\}$ & $d = 9$



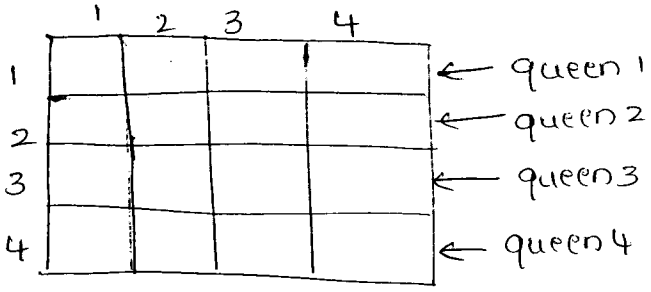
Soln:

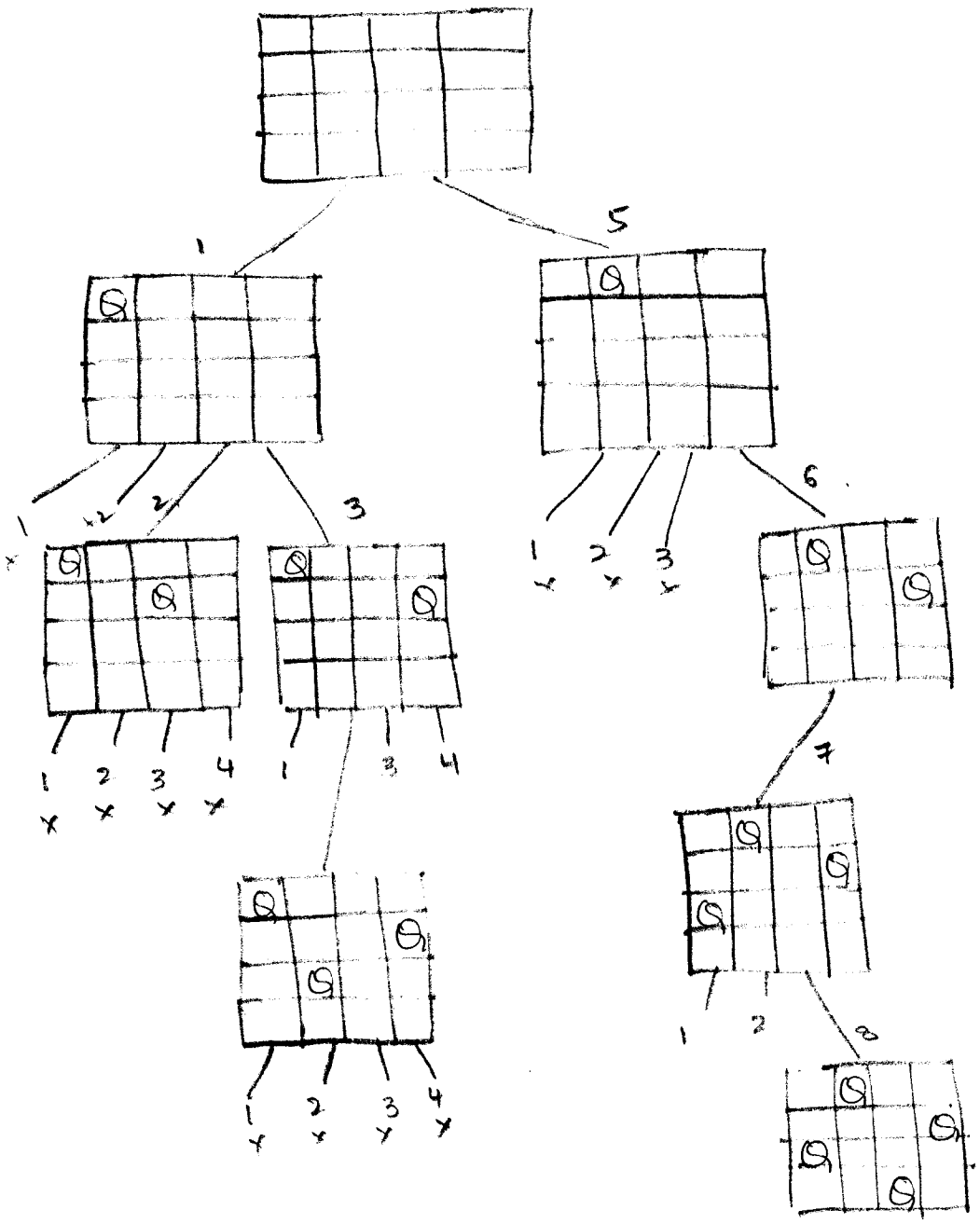
$S_1 = 1+3+6 = 9$ $\quad S_1 = \{1, 3, 6\}$

$S_2 = 1+8 = 9$ $\quad S_2 = \{1, 8\}$

10a The problem is to place n queens on an n-by-n chessboard so that no 2 queens attack each other by being in the same row in the same column or on same diagonal.

We start with the empty board & take place queen 1 in the first possible posit^n of its row which is in column 1 of row 1. Then we place queen 2 after trying unsuccessfully column 1 & 2 in the first acceptable position for it, square (2,3) the square in row 2 & column 3. This proves to be a dead end because there is no acceptable position for given queen 3. So the algorithm backtracks & put queen 2 in the next possible position at (2,4). The queen 3 is placed at (3,2) which proves to be another dead end. The algorithm then backtracks all the way to queen 1 & moves it to (1,2). Queen 2 then goes to (2,4) queen 3 to (3,1) & queen 4 to (4,3)





10b.

Item	weight	value	V/w
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Capacity of knapsack
$M = 10$.

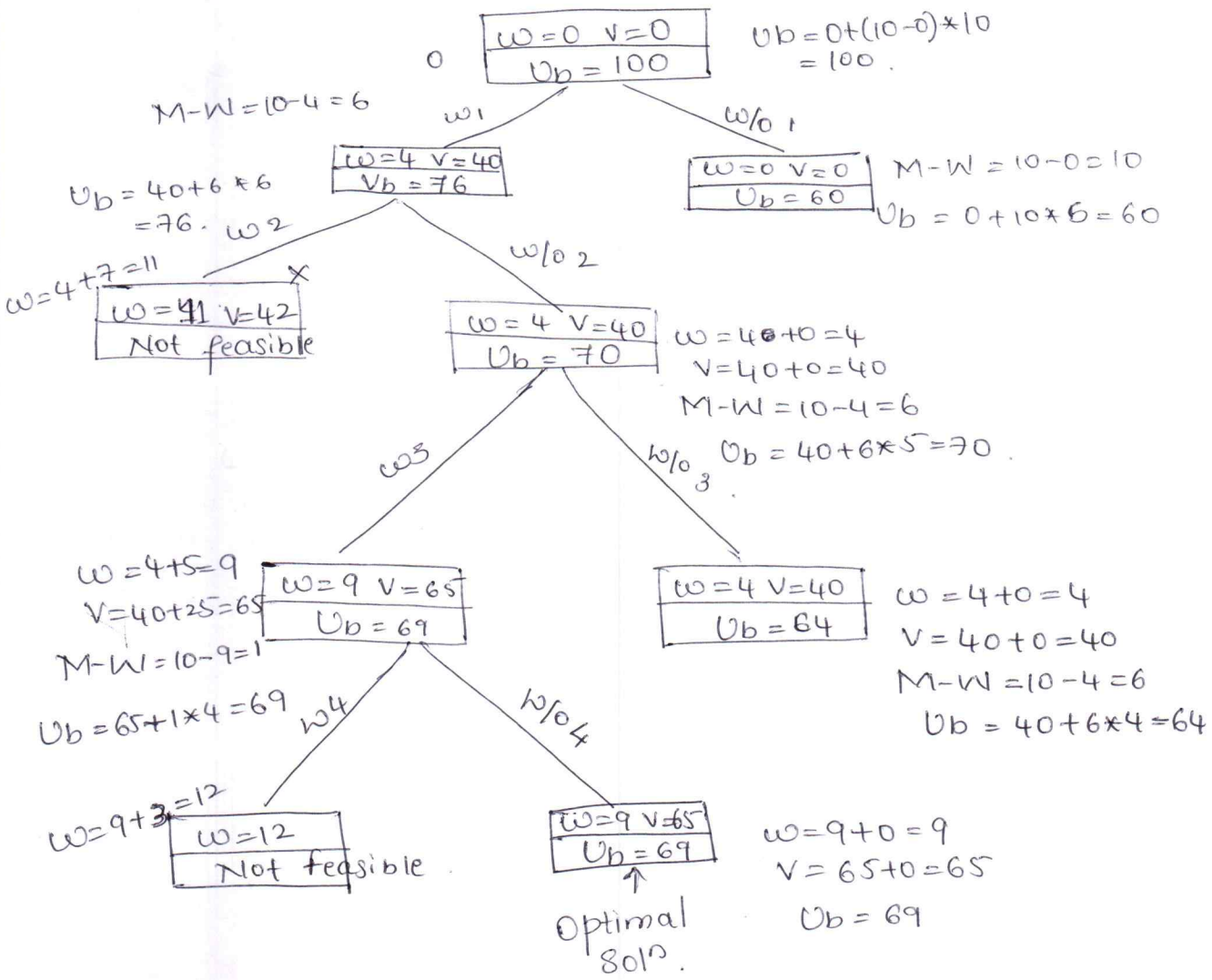
The upper bound can be calculated by using formula.

$ub = v + (M - w)(v_{i+1}/w_{i+1})$

w - Total weights of all the objects placed into the knapsack

v - Total profit or value of all objects placed into the knapsack.

State Space trees.



Solution:- {1,3}

Ub = 69.

Practical
7/7/25
HOD
CSE (AI & ML)
KLS Vishwanathrao Deshpande
Institute of Technology, Haliyal