Course Code:- MCS-031
Course Title:- Design and Analysis of Algorithm
Assignment No:- MCA(III)-031/Assignment/2019-20

Q.1. Enumerate five important characteristics of an Algorithm, and discuss any five well-known techniques for designing algorithm to solve the problems. State Travelling Sales Persons Problem. Comment on the nature of ~~solving~~ solution to the problem.

Ans:- The five important characteristics of an Algorithm are as follows:-

(i) Fitness:- An algorithm must terminate after ~~the~~ a finite number of steps and further each steps must be executable in finite amount of time. In order to establish a sequence of steps as an algorithm, it should be established that it terminates on all allowed inputs.

(ii) Definiteness:- Each step of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case. ~~Through~~

(iii) Inputs:- An algorithm has zero or more, but only finite, number of inputs.

(iv) Output:- An Algorithm has one or more outputs. The requirement of at least one output is obviously essential, because, otherwise we can't know the answer/solution provided by the algorithm.

(v) Effectiveness:- An algorithm should be effective. This means that each of the operation to be performed in an algorithm must be sufficiently basic that it can, in principle, be done exactly and in a finite length of time, by a person using pencil and paper. It may be noted that the 'FINITENESS' condition is a special case of 'EFFECTIVENESS'.

Algorithm Design Techniques:-

The following is a list of several popular design approaches:-

1. **Divide and Conquer Approach:-** It is a top-down approach. The algorithms which follows the divide & Conquare techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. **Greedy Technique:-** Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3. Dynamic Programming :- Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to Optimization problems.

4. Randomized Algorithms :- A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

5. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

Travelling Sales Person Problem:

The travelling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one and to return to the same city. The challange of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are $X_1, X_2, \ldots X_n$ where cost C_{ij} denotes the cost of travelling from city X_i to X_j. The travelling salesman problem is to find a route starting and ending at X_1 that will take in all cities with the minimum cost.

Example:- A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost.

Q. 2. Write recursive binary search algorithm and compare its run time complexity with the non recursive binary search algorithm. Solve the recurrence

$$T(n) = 2T(n/2) + n \quad n > 2$$
$$= 1 \quad n < 2.$$

Ans:- **Recursive Binary Search Algorithm**

Given:

key :→ Pointer to key of unknown type.

arr :- Array of a definite pointer type (that is, you can use expressions such as *arr[inx]).

Size :- The number of element in arr.

Cmp_proc :- Pointer to a caller-written function that can compare key to a value extracted from an element of arr. It is called like:

Cmp_proc (key, arr[inx])

and returns:

- 0 if key matches the value extracted from arr[inx];
- -1 if key is less than the value extracted from arr[inx]; or
- 1 if key is greater than the value extracted from arr[inx]

Returns:

The matching element, if found otherwise NULL.

The algorithm :-

1. Find the midpoint of the array; this will be the element at arr[size/2]. The midpoint divides the array into two smaller arrays : the lower half of the array consisting of elements of 0 to midpoint -1, and the upper half of the array consisting of elements midpoint to size -1.

2. Compare key to arr[midpoint] by calling the user function cmp_proc.

3. If the key is a match, return arr[midpoint]; Otherwise

4. If the array consists of only one element return NULL, ~~including~~ indicating that there are no match; Otherwise

5. If the key is less than the value extracted from arr[midpoint] search the lower half ~~to~~ of the array by recursively calling search; Otherwise

6. Search the upper half of the array by recursively calling search.

Difference b/w time complexity of recursive & non-recursive binary search algorithm :-

Recursive :- Time complexity of recursion can be found by finding the value of the nth recursive call in term of the previous calls. Thus, finding the destination case, and solving in terms of the base case gives us an idea of the time complexity of recursive equation.

For example:- Consider the recurrence

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

we guess the solution as,

$T(n) = O(n \log n)$. Now we use induction to prove our guess.

We need to provide that,

$T(n) <= cn\log n$. we can assume that it is true for values smaller than n.

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

$$<= cn/2 \log(n/2) + n$$

$$= cn\log n - cn\log 2 + n$$

$$= cn\log n - cn + n$$

$$<= cn\log n$$

NON-Recursive:- Time complexity of non-recursive binary search algorithm can be found by finding the number of cycles being repeated inside the loop.

Q.3. Derive the principle of optimality for multiplication of matrix chain. Compute the optimal number of scalar multiplication required to multiply the following matrices:.

A1 of order 30 * 35.
A2 of order 35 * 15.
A3 of order 15 * 5.

Ans:. Principle of Optimality:. A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.

In respect of multiplication of matrices, we recall the following facts:-

(i) Matrix multiplication is a binary operation, i.e., at one time only two matrices can be multiplied.

(ii) However, not every given pair of matrices may be multiplicable. If we have to find out the product M_1, M_2 then the order of the matrices must be of the form $m \times n$ and $n \times p$ for some positive integers m, n and p. Then matrices M_1 and M_2, in this order, are said to be multiplication-compatible. Number of scalar multiplications required is mnp.

(iii) Matrix multiplication is associative in the sense that if M_1, M_2 and M_3 are three matrices of order $m \times n$, $n \times p$ and $p \times q$ then the matrices

$(M_1 M_2) M_3$ and $M_1(M_2 M_3)$ are defined.

$(M_1 M_2) M_3 = M_1(M_2 M_3)$

and the product is an $m \times n$ matrix.

(iv) Though, for three or more options matrices, matrix multiplication is associative, yet the number of scalar multiplications may very significantly depending upon how we pair the matrices and their product matrices to get the final product.

Let's solve the multiplication of given matrices, for better explanation:-

A_1 of order $30 * 35$
A_2 of order $35 * 15$
A_3 of order $15 * 5$

Because, if A_1 is 30×35, A_2 is 35×15, and A_3 is 15×5 matrix, then the number of scalar multiplications required for $(AB)C$ or $(A_1A_2)A_3$ is:-

$30 \times 35 \times 15 = 15750$ (for (A_1A_2) which of order 30×15)
$(+) 30 \times 15 \times 5 = 2250$ (for product of (A_1A_2) with A_3)

Total = 18000 scalar Multiplication

On the other Hand, no. of scalar Multiplications for $A_1(A_2A_3)$ is:-

$35 \times 15 \times 5 = 2625$ (for (A_2A_3) which is of order 35×5)
$(+) 30 * 35 \times 5 = 5250$ (for product of with A_2A_3)

Total = 7875 scalar multiplication.

Q.4. Write Selection sort Algorithm. Use it to sort the list 90, 42, 41, 120, 160, 50. Calculate the complexity of the selection sort algorithm in best case, average case, and worst case.

Ans: The following steps constitute the selection sort algorithm:

Step 1:- Create a variable MAX to store the maximum of the value scanned upto a particular stage. Also create another variable say MAX-POS which keeps track of the position of such maximum values.

Step 2:- In each iteration, the whole list/array under consideration is scanned once to find out the current maximum value through the variable MAX and to find out the position of current maximun through MAX-POS.

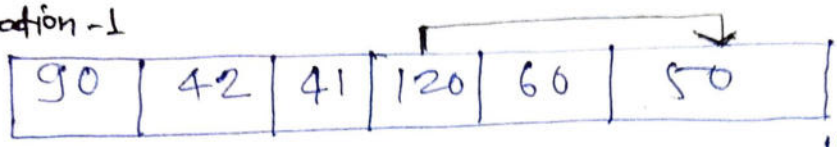
Step-3 :- At the end of an iteration, the value in last position in the current array and the (maximum) value in the position MAX-POS are exchanged.

Step 4:- For further consideration, replace the list L by L ~ {MAX} {and the array A by the corresponding subarray} and go to step 1.

Selection Sort:

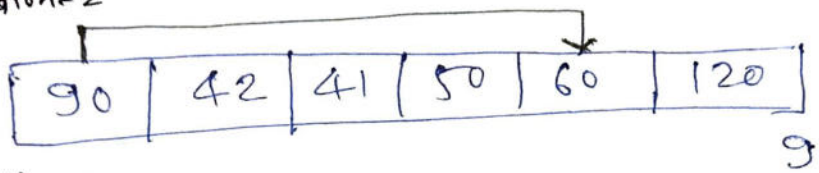
90, 42, 41, 120, 60, 50

Iteration-1



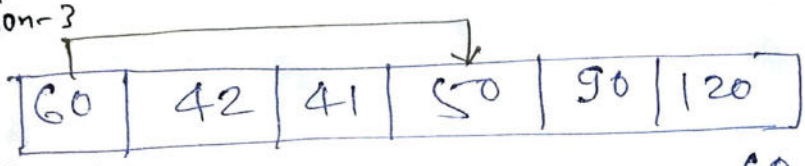
120 is largest

Iteration-2



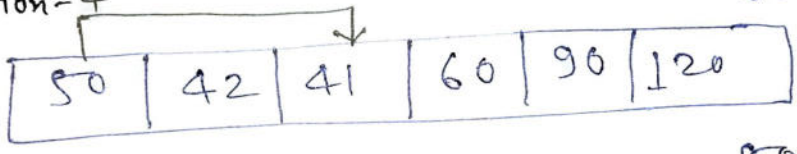
90 is largest

Iteration-3

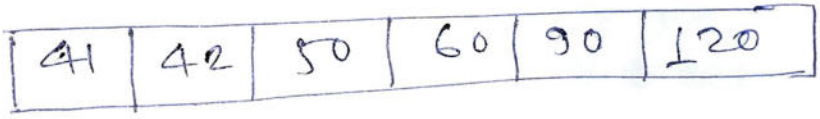


60 is largest

Iteration-4



50 is largest



Hence, no more iteration. The algorithm terminates. This complete the sorting of the given list.

Selection Sort Complexity

Worst-case performance ⟶ $O(n^2)$ comparisons, $O(n)$ swaps.

Best-case performance ⟶ $O(n^2)$ comparisons, $O(n)$ swaps.

Average-case performance ⟶ $O(n^2)$ comparisons, $O(n)$ swaps.

Worst-case space complexity :- $O(1)$ auxiliary.

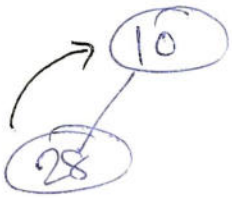
Q. 5. Sort the following elements using Heap sort:
10, 28, 46, 39, 15, 12, 18, 9, 56, 2. Show each step, while creating a heap and processing a heap. Also determine the Best case and worst case complexity of Heap sort algorithm. Prove that best case for bubble sort is worst case for heap sort.

List:- 10, 28, 46, 39, 15, 12, 18, 9, 56, 2

Ans:- Step:1 Construction of a Heap for the given list

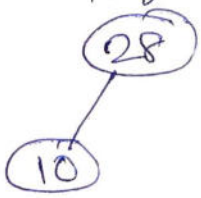
First we create a tree having the root as the only node with node-value 10 as shown in the list:-

Next value 28 will attached as left child of the root:-

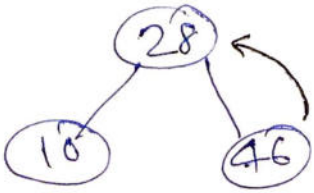


Here, the child node is greater than it's root node. Hence swaping is needed to satisfy the Heap property.

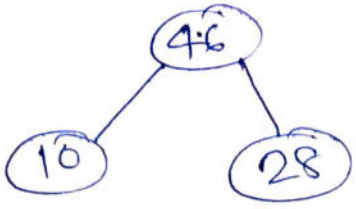
After swaping the tree will



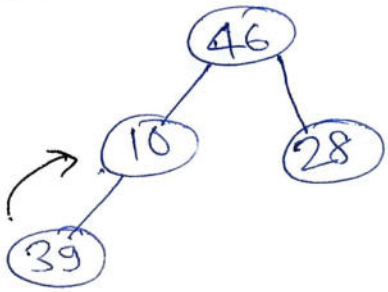
Next value 46 will added right child of the node:-



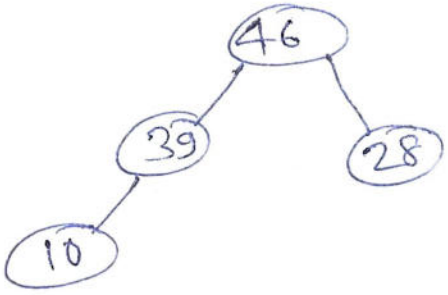
Again this tree disturb the Heap Property. Hence, swaping will perform:-



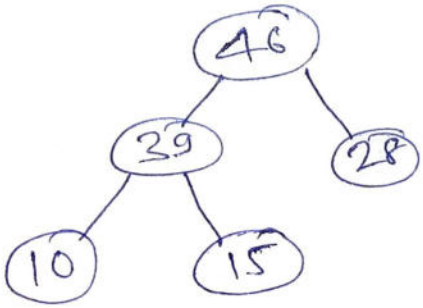
Next value 39 will attached as left child of 10:-



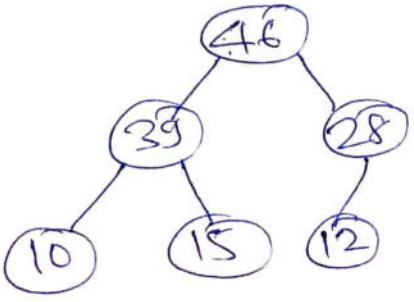
Swapping will perform Again, Because 39 > 10, and disturbing the Heap Property. After swaping :-



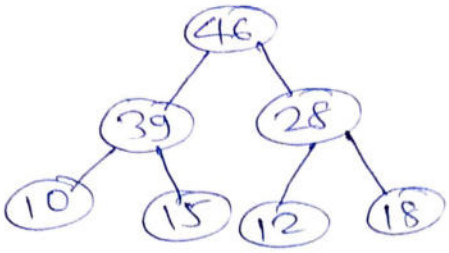
Next value 15 will attached as Right child of 39.



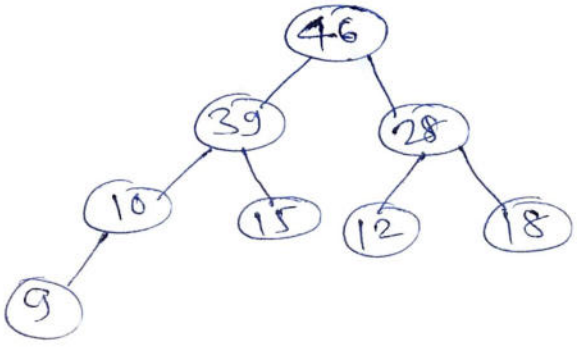
39 > 15, so it doesn't disturb the Heap property. So we will add the next value in the tree viz. 12 as left child of 28.



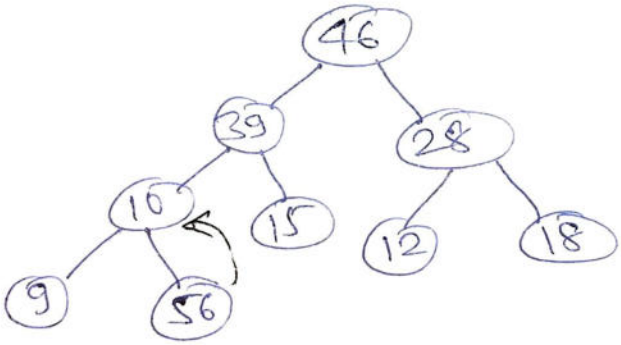
Again, 28 > 12, Heap property is not disturbed. So, we will add next value 18 as Right child of 28.



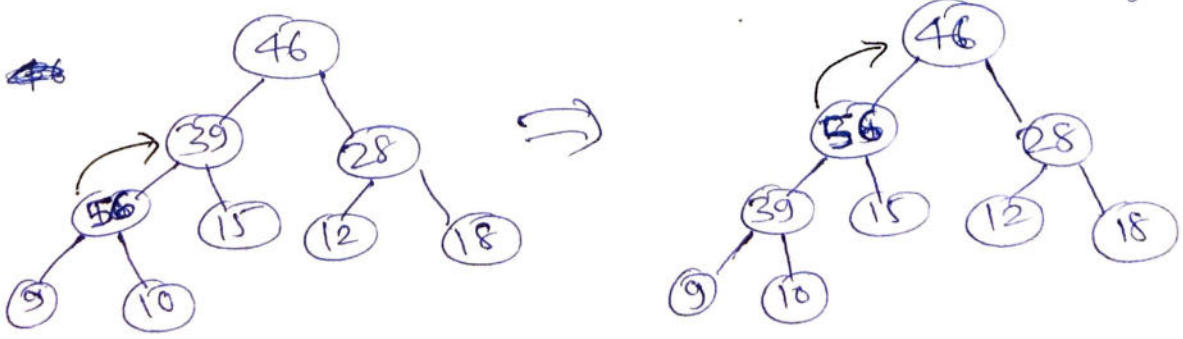
28 > 18, Heap property is not disturbed. So, we will add next value 9 as left child of 10.

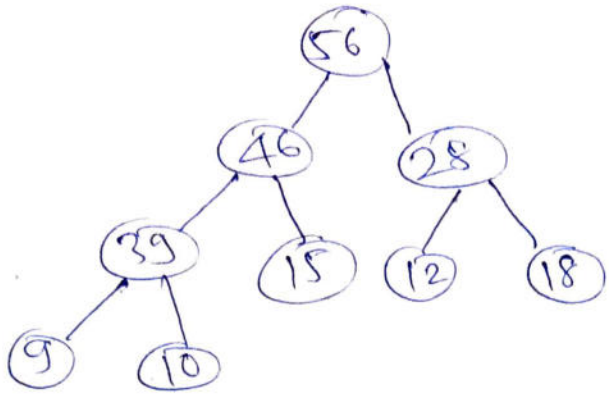


As the tree ~~sta~~ satisfies all the conditions of a Heap, we insert the next number 56 as Right child of 10.

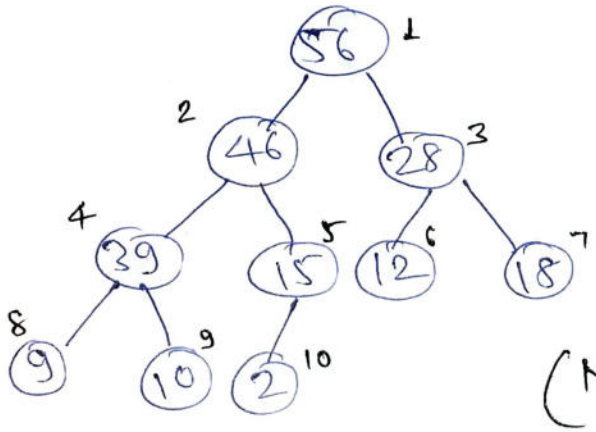


Here, Heap is disturbed. So will will do swaping again.





Now, Heap is satisfies, so, we add next value 2 as left child of 15.



(Numbers are allocating the array value.)

Finally, Heap is created. After Having constructed the Heap through Step 1, we perform Steps 2 & 3.

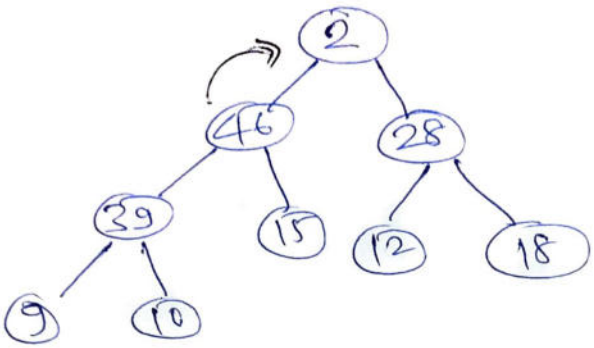
Step 2 & 3 :- Consists of a seqvence of actions viz

(i) deleting the value of the root,
(ii) Moving the last entry to the root and,
(iii) then readjusting the Heap.

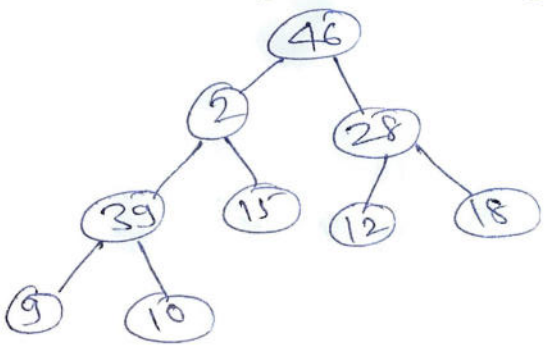
The root of a Heap is always the maximum of all the values in the nodes of the tree.

Here, the values are stored in the array says B[1,....,n] in which the values are to be stored after sorting in increasing order.

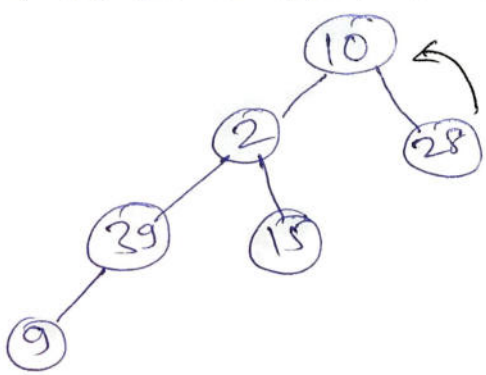
Next, value 2 is moved to the root and the node containing 2 is removed from further consideration to get the following binary tree, which is not a Heap.



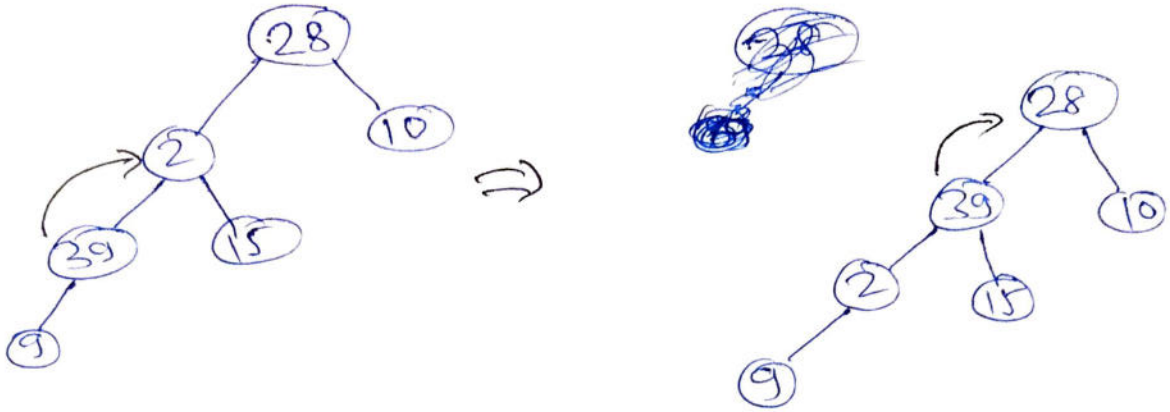
In order to restore the above tree as a Heap we perform swaping. And we get:-

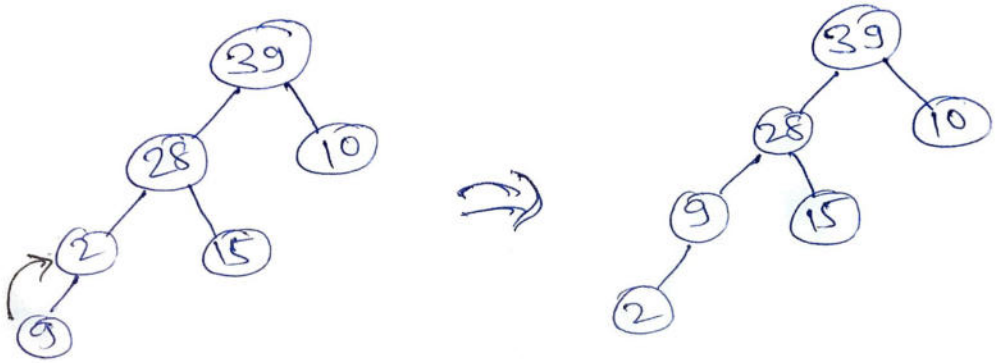


Again, 46 is copied to B[9] and 10, the last value of the tree is shifted to the root and last node is removed.

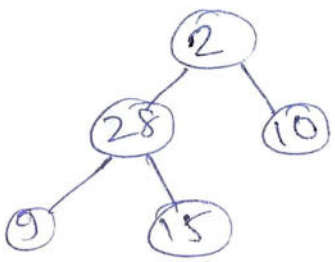


This is not a Heap. So we perform swaping operation to make it as a Heap.

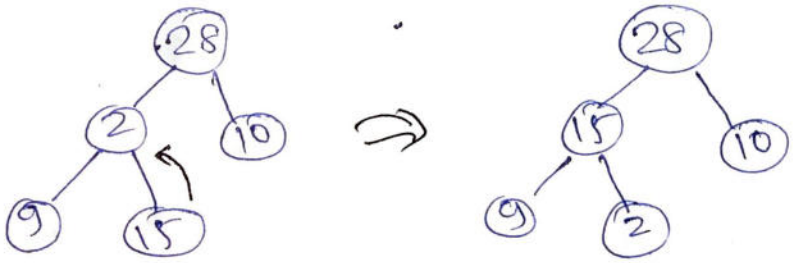




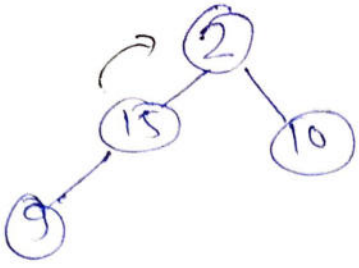
So, this is a Heap and now Again, we exchanged the value 39 with last value 2 and delete 39 from the tree.



Again we perform swapping to make it as a Heap.

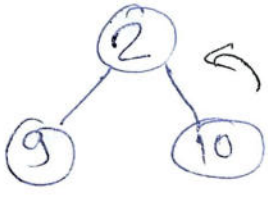


Next, 28 is copied ~~with~~ in B[5]. And the last node replaces the value in the root and the last node is deleted, to get the following tree which is not a Heap.

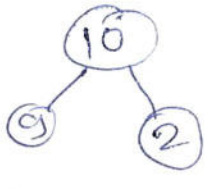


Again we swap the value to make it as a Heap.

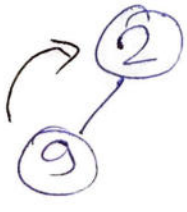
Next, 15 will copied with B[4] and the last node replaces the value in the root and the last node will deleted.



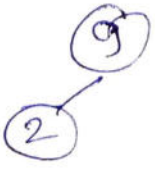
Again, we swap the value to make a Heap.



Next, 10 will copied with B[3] and the last node replaces the value in the root and the last node will delete.



Again, we swap the value to make it as a Heap.



Next, 2 will ~~copied~~ copy with B[2] and the last node replaces the value in the root and the last node will delete.

(2)

This value is copied in B[1] and the Heap algorithm terminates.

Best Case & Worst Case Complexity of Heapsort Algorithm :-

The worst case and Best case complexity for heap sort are both $O(n \log n)$. Therefore heap sort needs $O(n \log n)$ comparisons for any input array. Complexity of Heap Sort :-

$$O(n) \quad \text{(build (1,n) heap)}$$

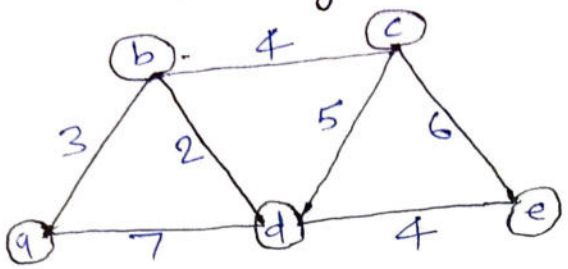
$$+ \sum_{i=1}^{n} O(\log i - \log 1) \quad \text{(build (1, j) heap)}$$

$$= O(n) + \sum_{i=1}^{n} O(\log i) \quad \text{(logarithm quotient rule)}$$

$$= O(n \log n) \quad \left(\sum_{i=1}^{n} \log i < \sum_{i=1}^{n} \log n = n \log n \right)$$

* Bubble sort has worst case average complexity both $O(n^2)$, where n is the no. of items being ~~sorted~~ sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other $O(n^2)$ sorting algorithm, such as insertion sort, ~~tend~~ to have better performance than bubble sort. Therefore bubble sort is not a practicle sorting algorithm when n is large.

Q. 6. Using Dijkshtra's algorithm, find the minimum distances of all the nodes from source node 'a' for the following Graph:

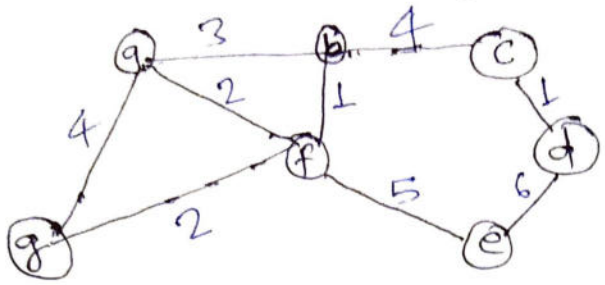


Ans:-

Step	Additional Node	S = Set-of-Remaining Nodes	Distances from source of b, c, d, e
Initialization	a	(b, c, d, e)	[3, ∞, 7, ∞]
1	b	(c, d, e)	[3, 3+4, 3+2, ∞]
2	d	(c, e)	[3, 3+4, 3+2, 3+2+4]
3	c	(e)	[3, 7, 5, 9]

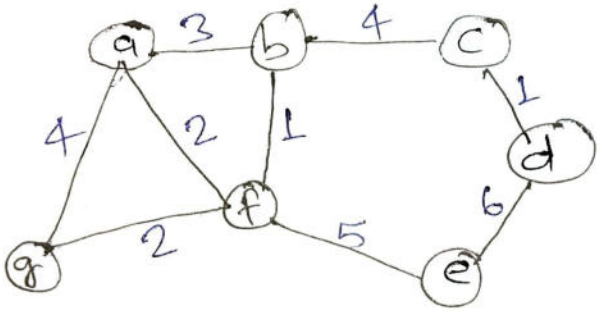
For minimum distance from a, the node b is directly accessed; the node c is accessed through b; the node d is accessed through b; and the node e is accessed through b and d.

Q.7. Obtain the minimum cost spanning tree for the following graph using Prim's algorithm



Obtain the DFS and BFS tree for the graph given considering node "a" as root node.

Ans:- Let's solve the above ~~tree~~ Graph using Prim's algorithm to find a minimum cost spanning tree :-



Initially :-

$V_T = (a)$

$E_T = \phi$

where, V_T = Set of Vertices.

E_T = Set of Edges.

1st Iteration :-

$V_T = (a, f)$

$E_T = ((a, f)$

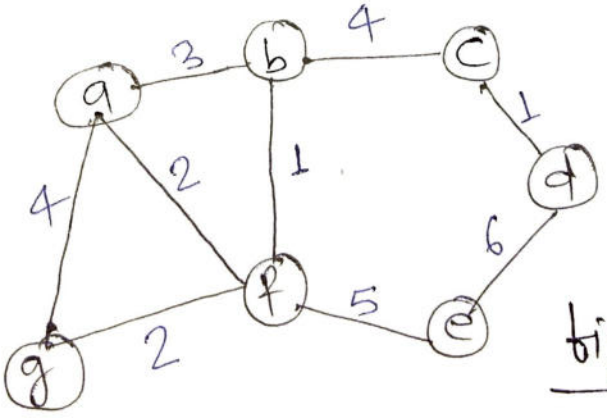


fig – 1

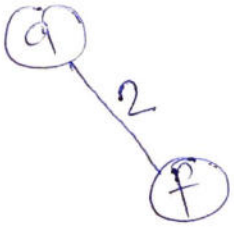


fig – 1.1

2nd Iteration :-

$V_T = (a, f, g)$

$E_T = ((a,f), (f,g))$

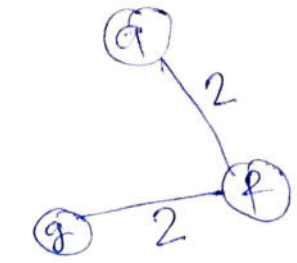


fig - 1.2

3rd Iteration :

$V_T = (a, f, g, b)$

$E_T = ((a,f), (f,g), (f,b))$

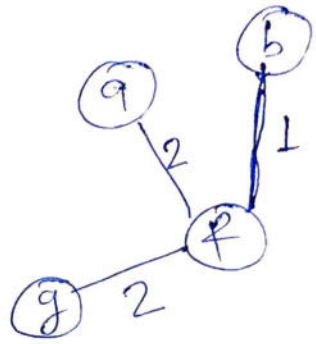


fig - 1.3

4th Iteration :-

$V_T = (a, f, g, b, c)$

$E_T = ((a,f), (f,g), (f,b), (b,c))$

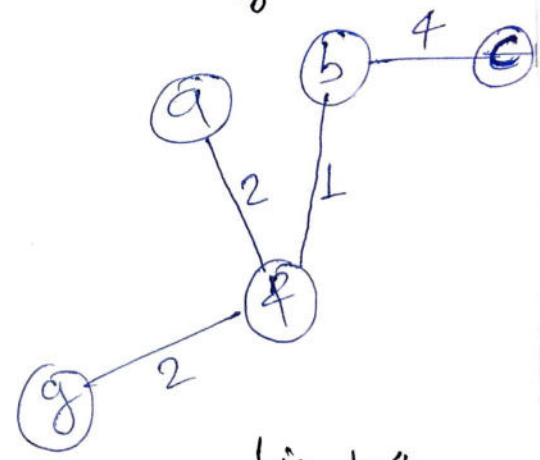


fig - 1.4

5th Iteration :-

$V_T = (a, f, g, b, c, d)$

$E_T = ((a,f), (f,g), (f,b), (b,c), (c,d))$

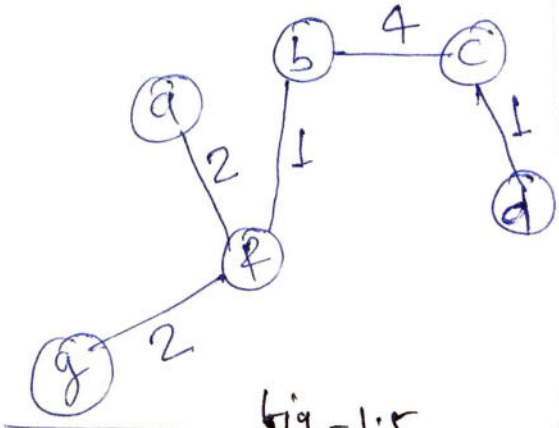


fig - 1.5

6th Iteration :-

$V_T = (a, f, g, b, c, d, e)$

$E_T = ((a,f), (f,g), (f,b), (b,c), (c,d), (f,e))$.

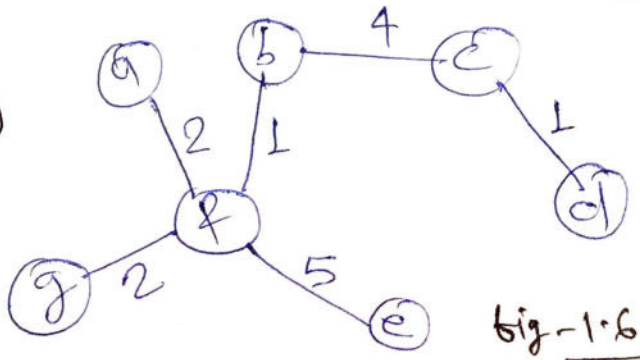
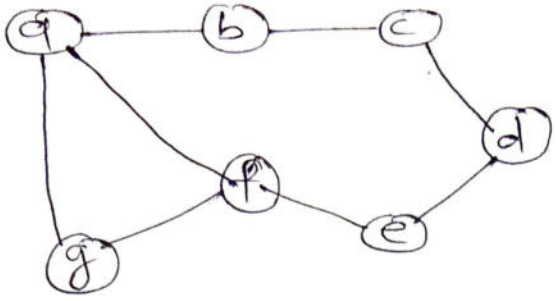
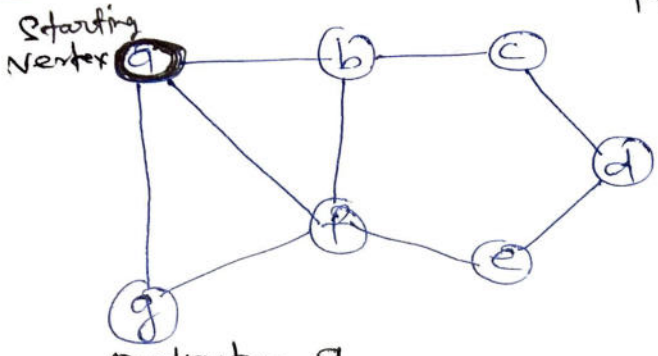


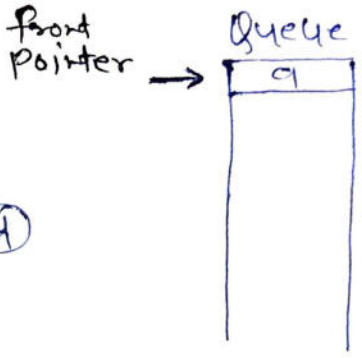
fig - 1.6

BFS for the given Graph:-



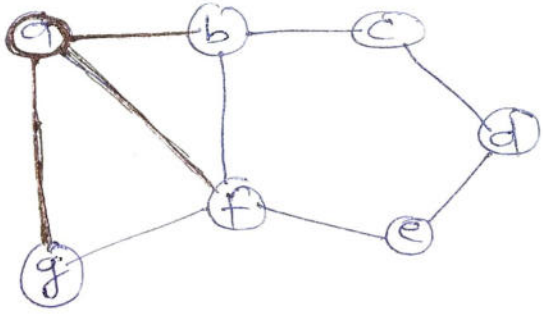
Step-1

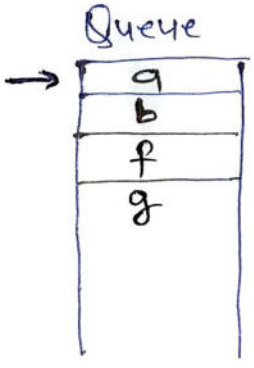




Output - a

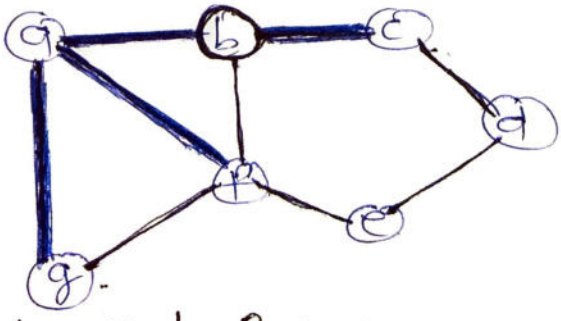
Step-2 Visit the adjucent of (a). Viz. (b), (f), (g)

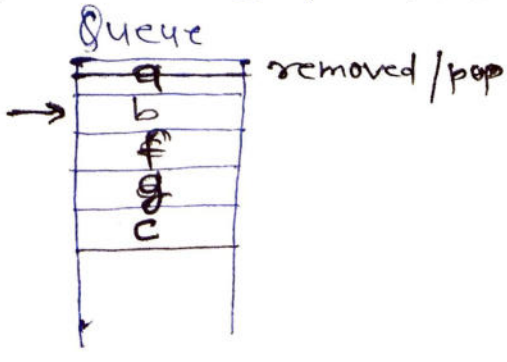




Output - a b f g

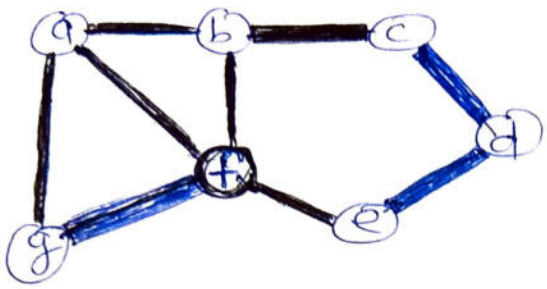
Step-3 All the adjucents of (a) are visited. So, now we shift the front pointer to (b) and check its adjucent unvisited vertices. And delete (a) from Queue.

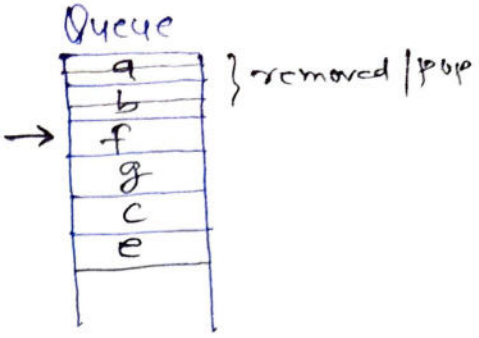




Output: a b f g c

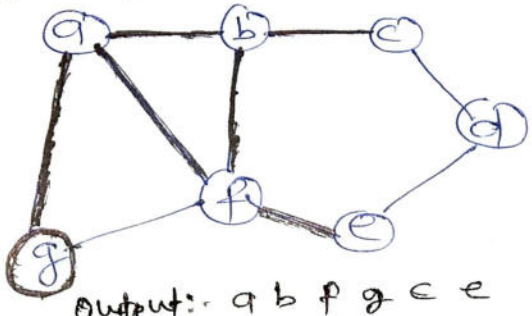
Step-4 All the adjucent vertices of (b) are visited. So, Shift the pointer to (f) and visit it's adjucent unvisited vertices. And remove (b) from Queue.



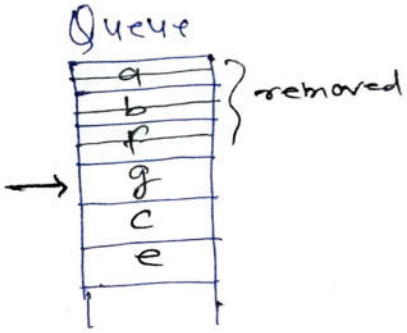


Output:- a b f g c e

Step-5 :- Shift the pointer to (g).

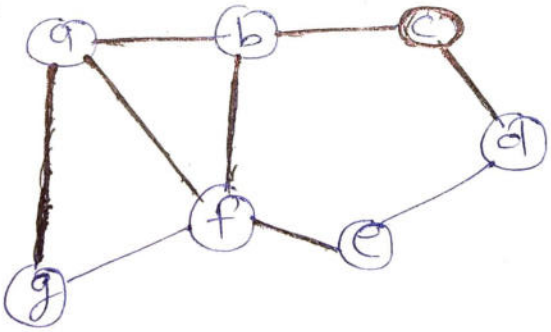


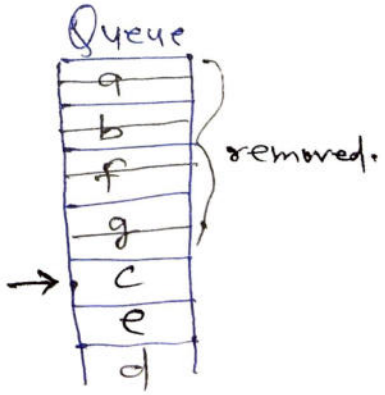
Output:- a b f g c e



We get, All the adjucent vertices of (g) are visited already. So, we remove the (g) from Queue and shift the pointer to (c) and visit it's unvisited vertices.

Step-6 :-

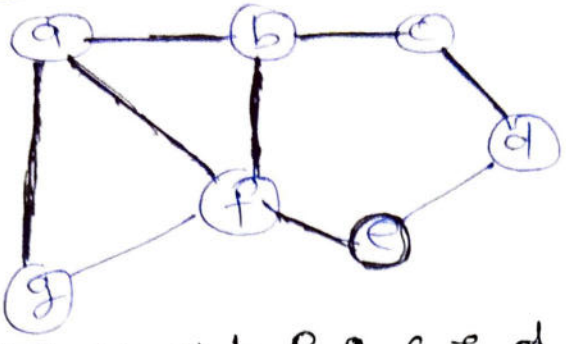


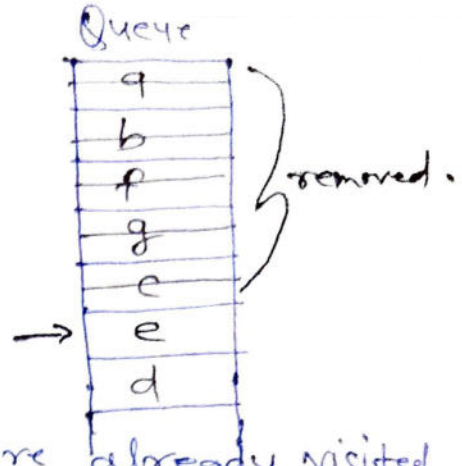


Output:- a b f g c e d

Now, all the adjucent vertices of (c) are visited. So, remove (c) from Queue and shift the pointer to (e) and visit it's unvisited vertices.

Step-7

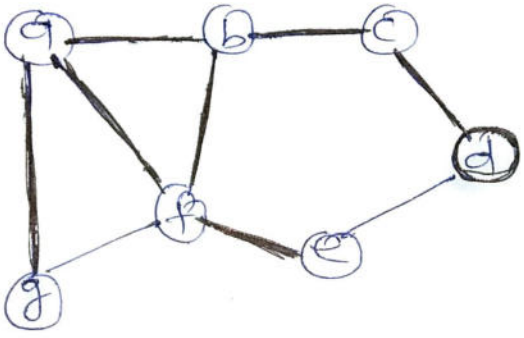


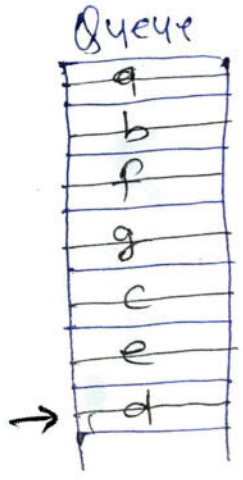


Output:- a b f g c e d

All the adjucent vertices of (e) are already visited so, we remove (e) from Queue and shift the pointer to (d) and ~~check~~ visit it's unvisited vertices.

Step-8



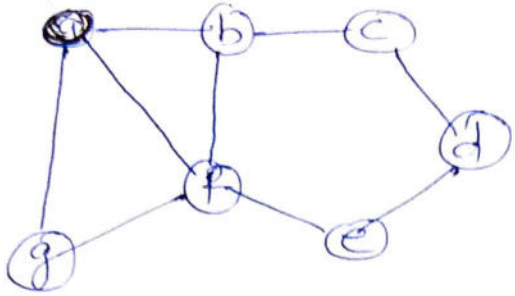


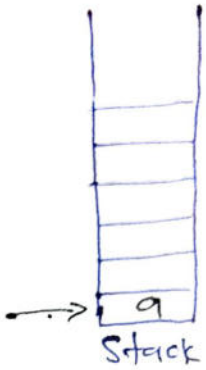
All the adjucent vertices of (d) are also visited. So, we remove (d) also from Queue and Queue will empty.

Output:- a b f g c e d

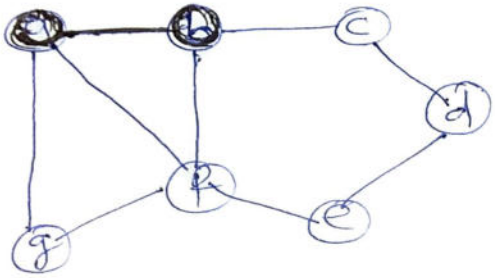
DFS for the given Graph:-

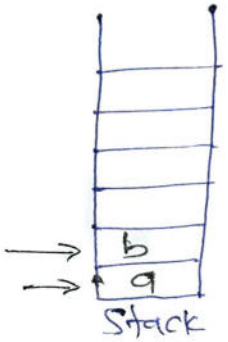
Step-1 Visit vertex (a)



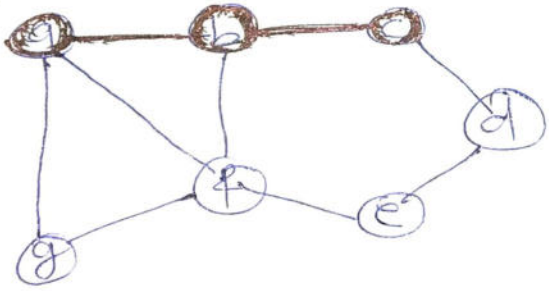


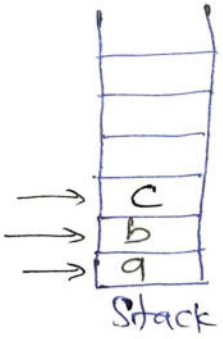
Step-2 :- Visit vertex (b)



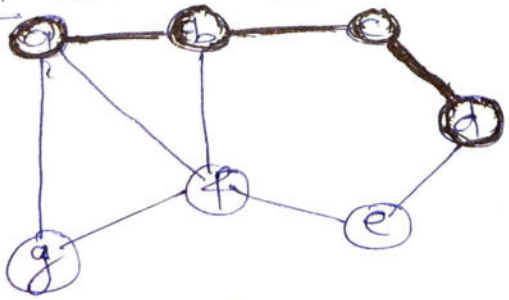


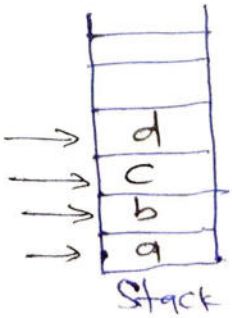
Step-3 :- Visit vertex (c)



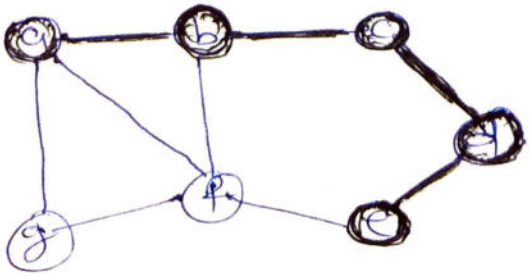


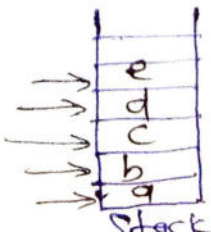
Step-4 Visit vertex (d)



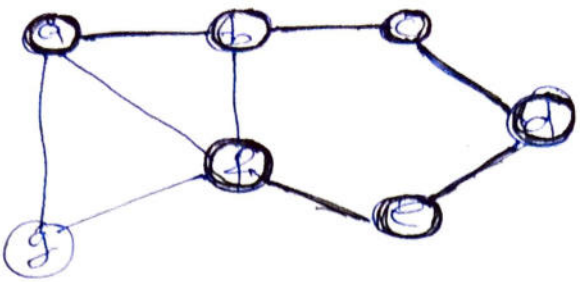


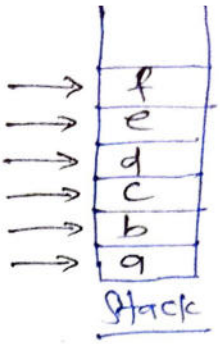
Step-5 Visit vertex (e)



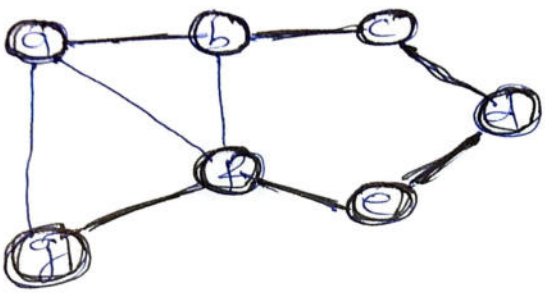


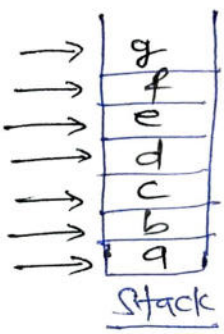
Step-6 Visit vertex (f)





Step-7

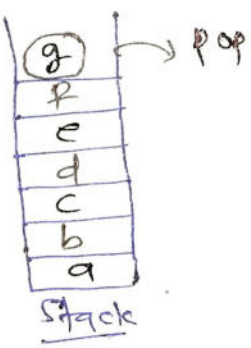




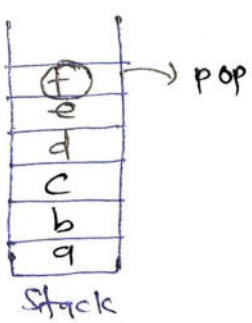
Output: a b c d e f g

Now no new vertex is adjacent to (g) viz unvisited. so, we backtrack / pop (g) from the stack. and do this step until we reached at Starting vertex.

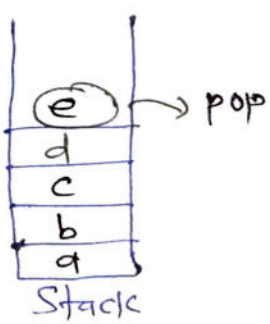
Step-8



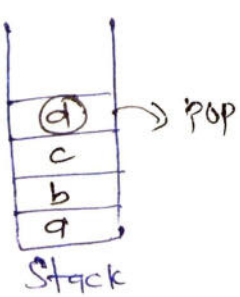
Step-9



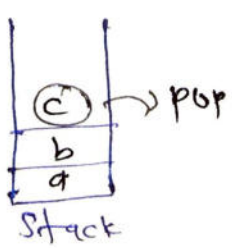
Step-10



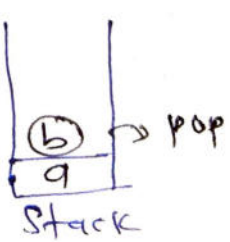
Step-11



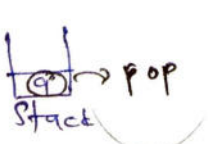
Step-12



Step-13



Step-14



Q.8. Explain the Chomsky's classification of grammers. What is an ambiguous grammer? How do you prove that a given grammer is ambiguous? Explain with an example. Write a context free grammer to generate palindromes of even length over the set of alphabets $\Sigma = \{a, b\}$.

Ans:- Chomsky classification for Grammer:-

As you have seen earlier, there may be many kinds of production rules. So, on the basis of production rules we can classify a grammer. According to Chomsky classification grammer is classified into the following types:

Type 0: This grammer is also called unrestricted grammer. As its name suggests, it is the grammar whose production rules and unrestricted.

All grammars are of type 0.

Type 1: This grammar is also called context sensitive grammar. A production of the form $xAy \rightarrow xay$ is called a type 1 production if $a \neq \Lambda$, which means length of the working string does not decrease.

A grammar is called type 1 grammar, if all of its productions are of type 1. For this, grammar $S \rightarrow \Lambda$ is also allowed.

The language generated by a type 1 grammar is called a type 1 or context sensitive language.

Type-2: The grammar is also known as Context free grammar. A grammar is called type 2 grammar if all the production rules are of type 2. A production is said to be of type 2 if it is of the form $A \rightarrow a$ where $A \in V$ and $a \in (V \cup \Sigma)^*$. The language generated by a type 2 grammar is to called context free language.

Type 3: A grammar is called type 3 grammar if all of its production rules are of type 3. (A production rule is of type 3 if it is form $A \rightarrow \Lambda$, $A \rightarrow a$ or $A \rightarrow aB$ where $a \in \Sigma$ and $A, B \in V$), i.e., if a variable derives a terminal or a terminal with one variable. This type 3 grammar is also known as, regular grammar. The language generated by this grammar is called, regular language.

* **Ambiguous Grammar**:- A grammar is said to be ambiguous if its language contains some string that has two different parse tree. This is equivalent to saying that some string has two distinct leftmost derivations or that some string has two distinct rightmost derivations.

For Example:- Suppose we define a set of arithmetic expressions by the grammar:

$$E \rightarrow a/b/E-E$$

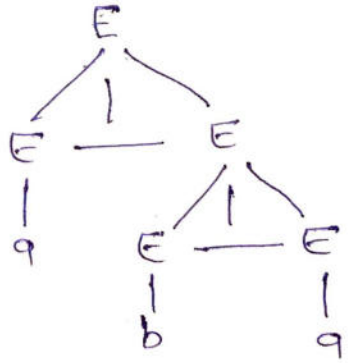


fig:- Parsetree

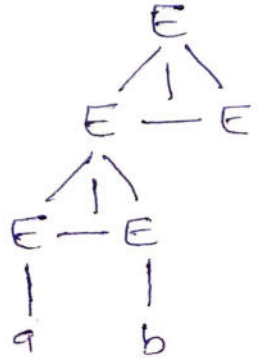


Fig:- Parse tree showing ambiguity

This is the parse tree for an ambiguous string.
The language of the grammar $E \rightarrow a/b/E-E$ contains strings like a, b, b-a, a-b-a, and b-b-a-b. This grammar is ambiguous because it has a string, namely, a-b-a, that has two distinct parse trees.

Since having two distinct parse trees mean the same as having two distinct left most derivations.

$E \Rightarrow E-E \Rightarrow a-E \Rightarrow a-E-E \Rightarrow a-b-E \Rightarrow a-b-a.$

$E \Rightarrow E-E \Rightarrow E-E-E \Rightarrow a-E-E \Rightarrow a-b-E \Rightarrow a-b-a.$

Context free grammar to generate palindroms of even length over the set of alphabets $\Sigma = \{a,b\}$:-

Let S be the starting symbol. Then the language of palindroms over the alphabet Σ has the grammar.

$S \rightarrow aSa/bSb/a/b/\Lambda.$

Q.9. What are Context free language? How they are different from context sensitive Languages? If L_1 and L_2 are context-free languages then, prove that $L_1 \cup L_2$ is context free language.

Ans:- Context-Free Language:- Since the set of regular language is closed under all the operations of union, concatenation, kleen star, intersection and complement. The set of context free language is closed under union, concatenation, Kleen Star only.

It is different from context sensitive language because, A type 1 grammar is also called context sensitive grammer and the language generated by a type 1 grammar is called context sensitive language.

A grammar is called type 1 grammar, if all of its productions are of type 1. For this, grammar $S \rightarrow \wedge$ is also allowed.

Proof of $L_1 \cup L_2$ is context-free language :-

This proof is similar to the Concatenation. If L_1 and L_2 are context-free languages, then each of them has a context-free grammar; call the grammars G_{1_1} and G_{12}. Our proof requires that the grammars have no non-terminals in common. So we shall subscript all of G_{1_1}'s non-terminals with a 1 and subscripts all of G_{12}'s non-terminals with a 2. Now, we combine the two grammars into one grammar that will generate the union of the two languages. To do this, we add one new non-terminal, S, and two new productions.

$$S \rightarrow S_1 \mid S_2$$

S is the starting non-terminal for the new union grammar and can be replaced either by the starting non-terminal for G_{1_1} or for G_{12}, thereby generating either a string from L_1 or from L_2. Since the non-terminals of the two original languages are completely different, and once we begin using one of the original grammars, we must complete the derivation using only the rules from that original grammar. Note that there is no need for the alphabets of the two languages to be the same.

Q.10. Compare Turing Machine and push down Automata. Construct a Turing Machine TM to accept all languages of palindromes on the set of alphabets $\Sigma = (a, b)$.

Ans:- Comparison of Turing Machine and Push down Automata are as follows:-

A PDA (Push down Automata) can only access the top of its stack, whereas a TM can access any position on an infinite tape. The infinite tape can't be simulated with a single stack, so a PDA is less computationally powerful. There are algorithms that can be programmed with a TM that can't be programmed with a PDA. An Automaton with access to two stacks rather than just one can simulate a TM and thus has equivalent computational power.

The language of any context-free grammar can be recognized by some non-deterministic PDA. The language of any formal grammar can be recognized by some TM. The latter are a strict superset of the former.

Let's Construct a Turing Machine TM to accept all languages of palindromes on the set of alphabets $\Sigma = (a, b)$:-

The required TM $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$ with

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, h\}$

$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$.

~~The next move function is given by~~

The proposed TM functions are as follows:-

In State q_0, at any stage if TM finds the black symbol then TM has found a palindrom of even length. Otherwise, it notes the symbol being read and attamts to match it with last non-blank symbol on the tape. If the symbol is 'a', the TM replaces it by #. goes to State q_1, in which it skips all a's and b's and on #, the TM from q_1 will go to q_3 to find a matching a in last non-blank symbol position. If a is found, TM goes to q_5 replace 'a' by #.

However, if b is found then TM has no more indicating the string is not a palindrome. However, if in state q_2 only #'s are found, then it indicates that the previous 'a' was the middle most symbol of the given string indicating palindrome of odd length.

Similar is the case when b is found in State q_0, except that the next state is q_2 in this case and roles of a's and b's are interchanged in the above argument

Q.11. Compare Non-Deterministic Finite Automata and Deterministic Finite Automata. Write the finite automata corresponding to the regular expression $(a+b)^*ab$. Also prove that for any set S of strings $S^* = (S^*)^* = S^{**}$.

Ans:- Some following points shows important differences b/w NFA and DFA :-

1. Deterministic Finite Automaton or DFA is a type of FA where in only one path is possible for any specific input to transit from its current state to the next state. There exists a unique transition for each input symbol. On the other hand, Non-Deterministic Finite Automaton or NFA refers to a type of FA wherein it is possible to have many paths for a given set of inputs to make their transition from their current state to the next states.

2. Empty String transition can't be used in DFA. Conversely, NFA Empty String transition is possible in NFA.

3. A DFA is best explained in the form of one machine and not as separate units for computing purposes. NFA is a collection of multiple little-sized units that are combined together to perform computation activities.

4. DFAs reject the string in case the termination state is other than the accepting state. On the other hand, NFA rejects the string when only all branches of the NFA are dead or don't accept the string.

5. There is just one transition state for every alphabetic symbol. Conversly, no user specifications are needed for making NFAs react in line to symbols.

* Regular expression $(a+b)^* ab$

$$= \{a, b\}^* \{ab\}$$

Now, we prove for any set S of strings prove that

$$S^* = (S^*)^* = S^{**}.$$

Proof:- We know that every word in S^{**} is made up of factors from S^*.

Also, every factor from S^* is made up of factors from S.

Therefore, we can say that every word in S^{**} is made up of factors from S.

First, we show $S^{**} \subset S^*$ ——— (I)

Let $X \in S^{**}$ ... Then $X = X_1 \ldots X_n$ for some $X_1 \in S^*$ which implies $S^{**} \subset S^*$

Next, we show $S^* \subset S^{**}$

$S^* \subset S^{**}$ ——— (II)

By above inclusions (I) and (II), we proved that

$S^* = S^{**}$

Q.12. Differentiate the following give suitable example for each :-

(a) NP- hard problem and NP complete problems
(b) Push Down Automata and Turing Machine.
(c) Decidable problems and Undecidable problems.
(d) Quick Sort and Randomized quick sort.
(e) Greedy Technique and Divide and conquer Technique.

Ans:- (a) NP - hard problem and NP Complete problems:-

NP - Complete problems:- A problem P or equivalently its language L_1 is said to be NP- complete if the following two conditions are satisfied :

(i) The problem L_2 is in the class NP.
(ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2.

Example:- 3-SAT. This is the problem wherein we are given a conjunction (ANDs) of 3-clause disjunctions (ORs), statements of the form.

(X_V11 OR X_V21 OR X_V31) AND (X_V12 OR X_V22 OR X_V32) AND... AND (X_V1n OR X_V2n OR X_V3n)

where each X_Vij is a boolean variable or the negation of a variable from a finite predefined list (X_1, X_2, ..., X_n).

It can be shown that every NP problem can be reduced to 3-SAT. The proof of this is technical and requires use of the technical definition of NP. This is known as Cook's theorem.

NP-Hard Problem:- A problem L is hard if only condition (ii) of NP-Completeness is satisfied. But the problem has may be so hard that establishing L as an NP-class problem is so far not possible.

However, from the above definitions, it is clear that every NP-complete problem L must be NP-hard and additionally should satisfy the condition that L is an NP-class problem.

Example:-

The halting problem is an NP-hard problem. This is the problem that given a problem P and input I, will it halt? This is a decesion problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

(b) Difference b/w Push Down Automata and Turing Machine:-

Push Down Automata:- Push down Automata is a branch of theory based computer science and it is a type of automation is used in theories that can't be calculated by computers. Push down automation is a less capable of than Turing Machines. Push down Automation reads a given string from left to right. It can Also manipulate a string.

The language of any context-free grammar can be recognized by some non-deterministic Push Down Automata.

Turing Machine:- A Turing Machine is a hypothetical computer used to demonstrate that there are a type of issues that no computers can solve regarless of whether it has boundless time and limitless memory. The basic turing Machine has a data memory composed of cells. A "head" is positioned on one cell and can read its current symbol, write a new symbol, step backward/forward one cell.

The language of any formal grammar can be recognized by some Turing Machine.

(c) Decidable problems and Undecidable Problems:-

Decidable Problems:- A problem is said to be decidable if we can always construct a corresponding algorithm that can answer the problem correctly. We can intuitively understand Decidable problems by considering a simple example. Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000. To find the solution of this problem, we can easily devise an algorithm that can enumerate all the prime numbers in this range.

Undecidable Problems:- The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as, Undecidable problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

We can understand Undecidable problems intuitively by considering Fermat's Theorem a popular Undecidable problem which states that no three positive integers a, b and c for any $n>2$ can ever satisfy the equation: $a^n + b^n = c^n$.

If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable value of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an Undecidable Problem.

(d) Quick Sort and Randomized Quick Sort:-

Quick Sort:- Like Merge sort, Quick Sort is also a Divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways:-

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partition is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Randomized Quick Sort: In Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array). Instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[p...r]. We do so by first exchanging element A[r] with an element chosen at random from A[p...r]. By randomly sampling the array p, ..., r, we ensure that the pivot element $x = A[r]$ is equaly likely to be any of the $r-p+1$ elements in the subarray. Because, we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average.

(e) **Greedy Technique V/S Divide & Conquer Technique:**

Greedy Technique: A Greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benifit. So the problems where choosing locally optimal also leads to a global solution are best fit for Greedy.

For example: Consider the ~~factorial~~ fractional knapsack problem. The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to global optimal solution because we allowed taking fractions of an item.

Divide & Conquer Techniques:.

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps:-

1. Divide:. Break the given problems into subproblems ~~and~~ of same type.

2. Conquer:. Recursively solve these subproblems.

3. Combine:. Appropriate combine the answers.

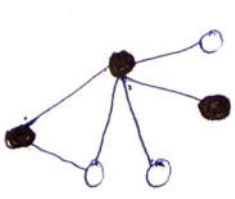
A classic example of Divide and Conquer technique is Merge Sort.

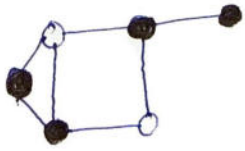
The divide-and-conquer techniques is the ~~best~~ basis of efficient algorithms for all kind of problems, such as sorting (e.g., Quicksort, merge sort), multiplying large numbers (e.g... the karatsuba algorithm), finding the closest pair of points, Syntactic analysis (e.g... top-down parsers), and computing the ~~distanc~~ discrete Fourier transform (FFT).

Q. 13. Discuss the following with suitable example for each:.

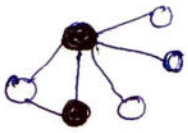
Ans:- (f) Vertex Cover Problem :- A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to atleast one vertex of the set. The problem of finding a minimum vertex cover is a classical optimization problem in Computer Science and is a typical example of an NP-Hard optimization problem that has an approximation algorithm. Its decision version, the vertex, the vertex cover problem, was one of Karp's 21 NP complete problems and is therefore a classical NP-complete problem in computational complexity theory.

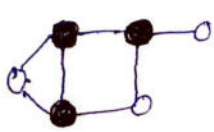
Formally, a vertex cover V' of an undirected graph $G = (V, E)$ is a subset of V such that ~~$\forall$~~ $uv \in E \Rightarrow u \in V' \vee u \in V'$, that is to say it is a set of vertices V' where every edges has at least one end point in the vertex cover V'. Such a set is ~~cover~~ said to cover the edges of G. The following figure shows two examples of vertex covers, with some vertex cover V' marked in Black





A minimum vertex cover is a vertex cover of smallest possible size. The vertex cover number T is the size of a minimum vertex cover, i.e. $T = |V'|$. The following figure shows examples of minimum vertex covers in the previous graphs.





(g.) Knapsac Problem:- The knapsack problem is a problem in combinatorial optimizati-on:- Given a set of items, each with a weight and a value, determine the no. of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

The problem often arises in resource allocation where there are financial constraints and is studied in fields such as combinatorics, computer science, complexity theory, cryptography, applied mathematics, and daily fantasy sports.

The most common problem being solved is the 0-1 knapsack problem, which restricts the no. x_i of copies of each kind of item to zero or one. Given a set of n items numbered from 1 upto n, each with a weight w_i and a value v_i, along with a maximum weight capacity W,

$$\text{maximize} \sum_{i=1}^{n} v_i x_i \qquad \text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and}$$

$$x_i \in \{0, 1\}.$$

Here x_i represents the no. of instances of item i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

(h) Strassen's algorithm :- The Strassen's algorithm, is an algorithm for matrix multiplication. It is faster than the ~~sat~~ standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithms for extremely large matrices.

Strassens algorithm works for any ring, such as plus / multiply, but not all semirings, such as min-plus or boolean algebra, where the naive algorithm still works, and so called combinatorial matrix multiplication.

For example:- Given two n by n matrices A and B, for multiplication :-

Split A and B into four matrices each:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Then calculate the following ten matrices:-

$S_1 = B_{12} - B_{22}$

$S_2 = A_{11} + A_{12}$

$S_3 = A_{21} + A_{22}$

$S_4 = B_{21} - B_{11}$

$S_5 = A_{11} + A_{22}$

$S_6 = B_{11} + B_{22}$

$S_7 = A_{12} - A_{22}$

$S_8 = B_{21} + B_{22}$

$S_9 = A_{11} - A_{21}$

$S_{10} = B_{11} + B_{12}$

Using the derived matrices above, calculate the following seven matrices recursively:

$$P_1 = A_{11} \cdot S_1$$
$$P_2 = S_2 \cdot B_{22}$$
$$P_3 = B_3 \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4$$
$$P_5 = S_5 \cdot S_6$$
$$P_6 = S_7 \cdot S_8$$
$$P_7 = S_9 \cdot S_{10}$$

Finally, although the algebra is omitted here, one can easily confirm that $C = A$

- B can be constructed in the following way:

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

(i) Dynamic programming:- Dynamic programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic programming. The idea is to simply store the results of subproblems, so that we don't have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

For example:- If we write simple recursive solution for Fibonacci Series, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

```
int fib (int n)
  {
    if (n <= 1)
      return n;
    return fib(n-1) + fib(n-2);
  }
```

Recursion : Exponential

```
f[0] = 0;
f[1] = 1;
for(i=2; i<=n; i++)
  {
    f[i] = f[i-1] + f[i-2];
  }
  return f[n];
```

Dynamic Programming : Linear.