IARE '24 DISCORD TEAM

II-II

DAA MODULE 1 SOLUTIONS

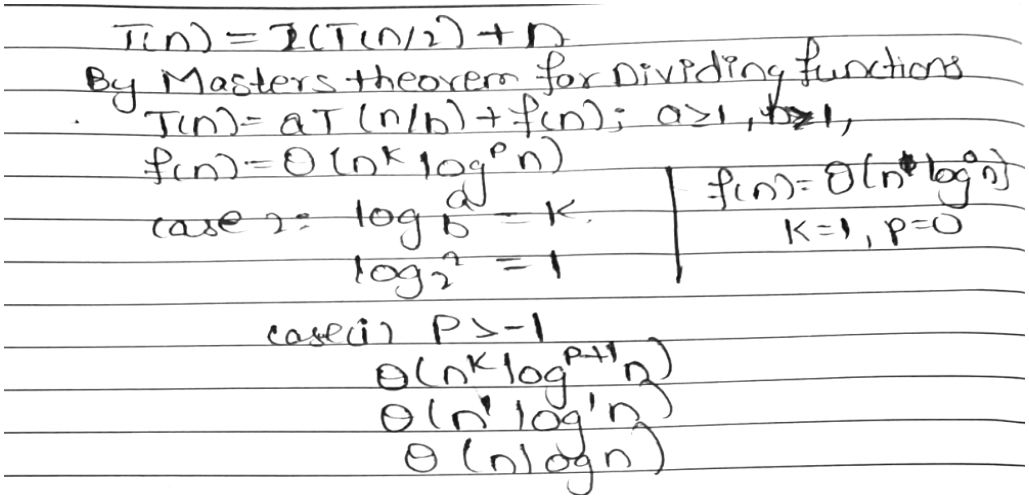
NEHANJALI • SHRESHIKA • UJJWAL • NISHANT • PRANAV

INTRO TO DESIGN AND ANALYSIS OF ALGORITHMS

DAA MODULE 1

PART A

1.Solve the following recurrence relation T(n)=2 T(n/2) + n, and T(1)=2.



2. Solve the following recurrence relation T(n) = 7T(n/2)+cn2

$T(n) = 7T(n/2) + cn^2$

Comparing with Master's theorem, we get :-

$T(n) = aT(n/b) + f(n)$

$\therefore a=7, b=2$

$T(n) = n^{\log_a^b}[u(n)]$

$= n^{\log_2^7}[u(n)]$

$= n^{2.807}[u(n)]$

$= n^{2.807} \cdot O(1)$

$= n^{2.807}$

$\therefore T(n) = n^{2.807}$

Value of $U(n)$ depends on $h(n)$

$h(n) = \frac{f(n)}{n^{\log_a^b}}$

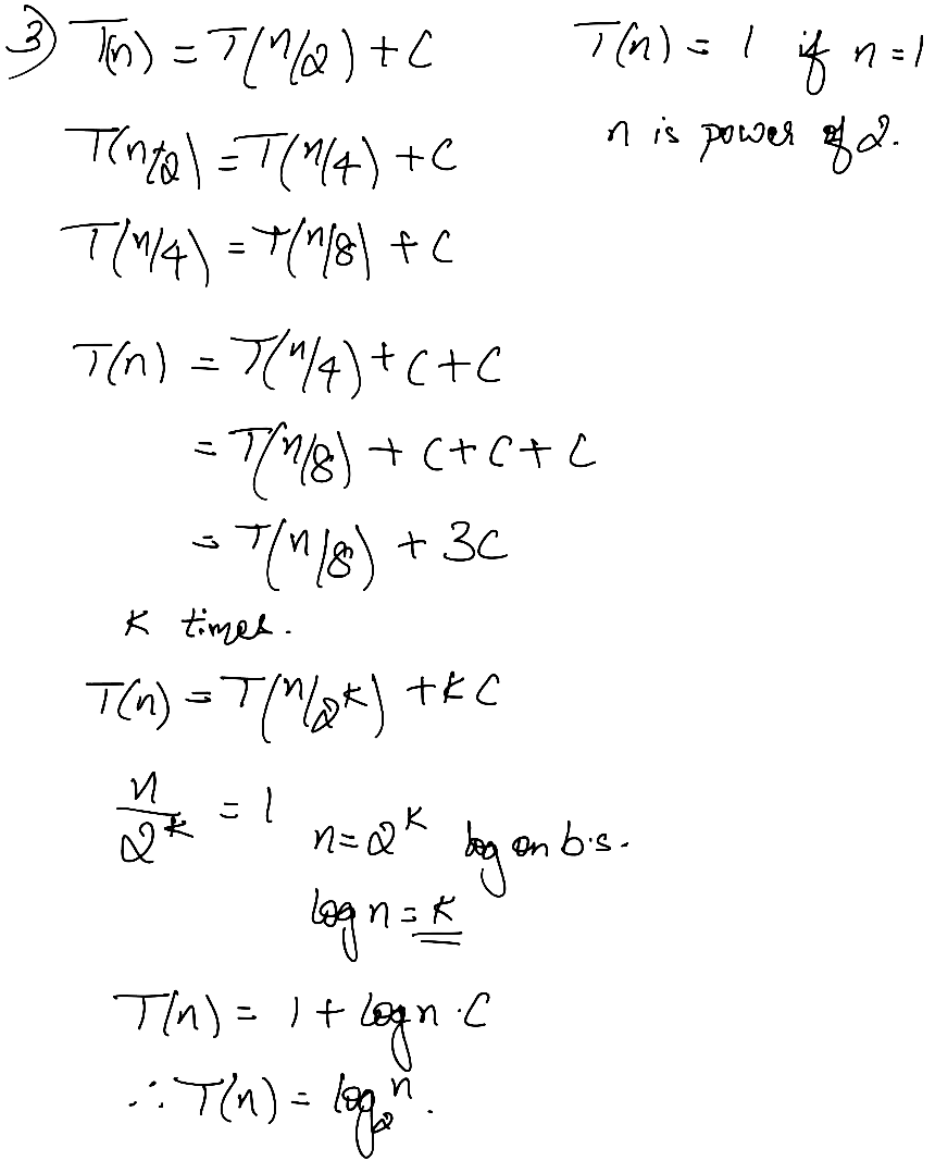
$= \frac{cn^2}{n^{2.807}} = cn^{-0.807}$

Since n^r is negative,

i.e. $r < 0$.

$U(n) = O(1)$

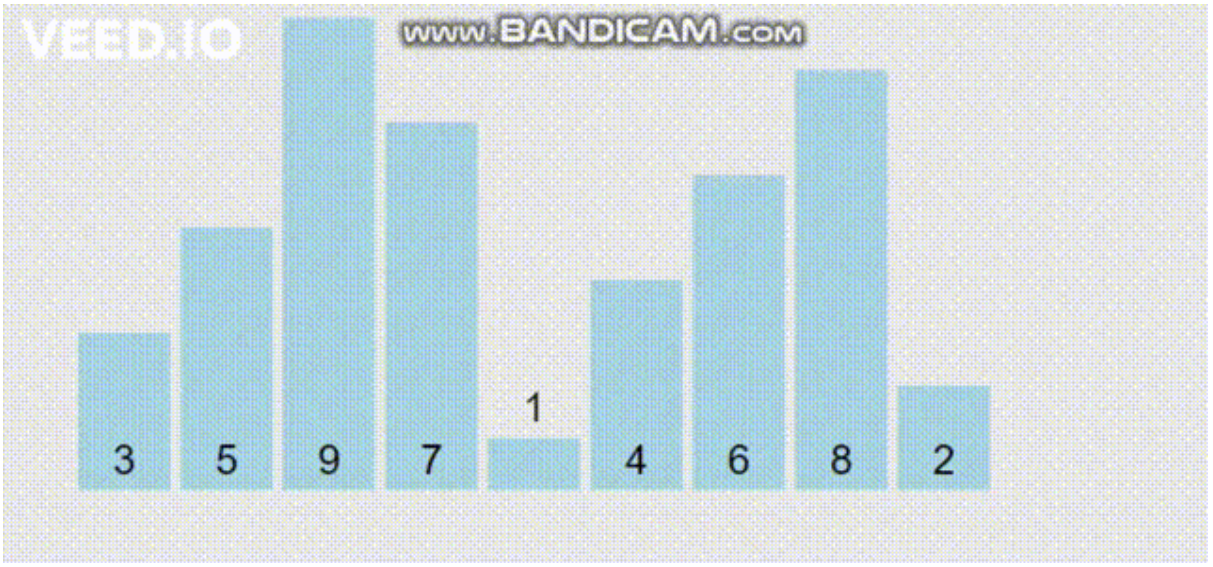
3. Solve the recurrence relation T(n)=T(1), n=1 T(n)=T(n/2) + c, n¿1 and n is a power of 2.



4. Apply a quick sort algorithm and simulate it for the following data sequence: 3 5 9 7 1 4 6 8 2.

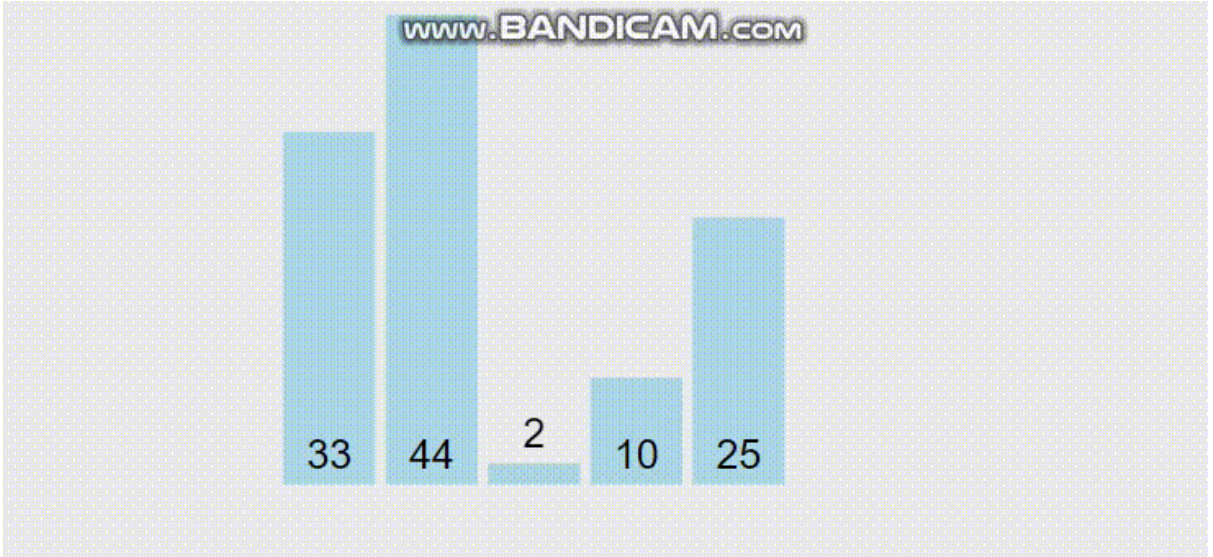
```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array, leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)
partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
```

```
  storeIndex <- leftmostIndex - 1
 for i <- leftmostIndex + 1 to rightmostIndex
 if element[i] < pivotElement
    swap element [i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```



5. Identify the tracing steps of merge sort and quicksort and analyse the time complexity for the following data: 33, 44, 2, 10, 25

Quick Sort:



Time Complexity:

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

Time Complexity:

Worst Case Time Complexity [Big-O]: **O(n*log n)**

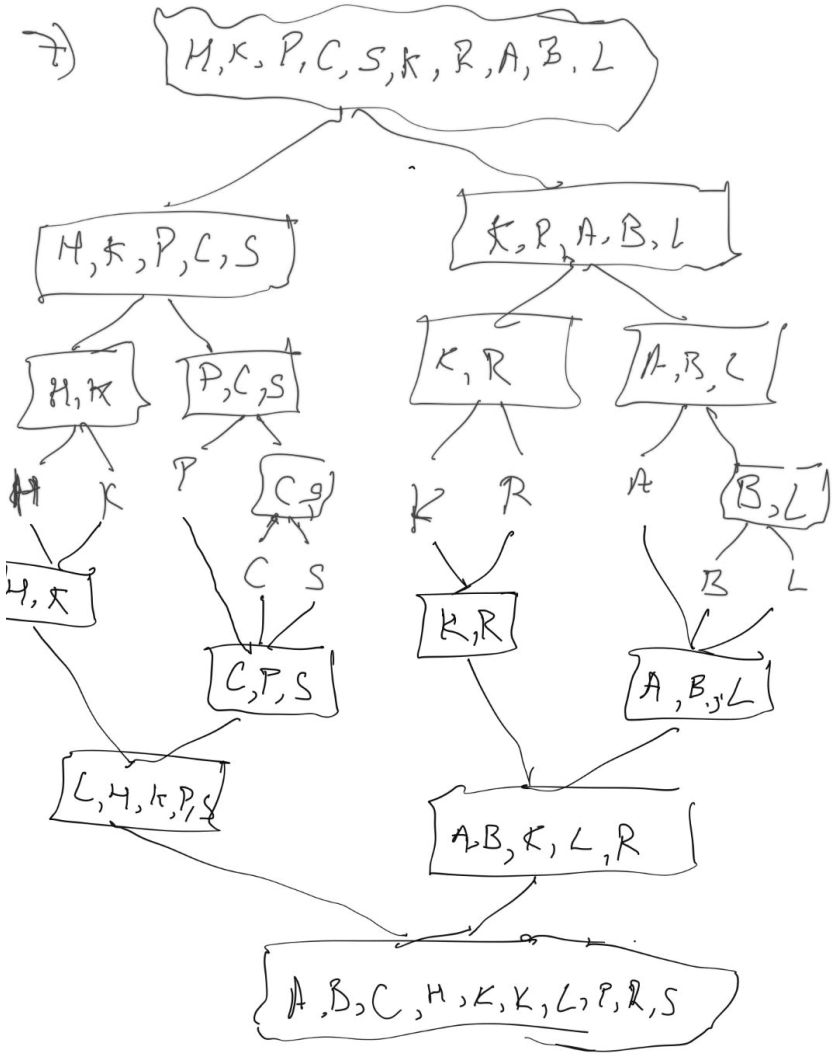
Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n)**

6. Organise the steps in merge sort to arrange following data in non-decreasing order 1,2,5,6,9,8,7

7. Organise merge sort on following letters H, K, P,C,S,K,R,A,B,L



8. Explain Strassen's method outperforms the traditional matrix multiplication method. How many multiplication operations are required during multiplication of two matrices with size of 32 x 32 in Stressen's method.

Using the traditional method, two matrices (X and Y) can be multiplied if the order of these matrices are $p \times q$ and $q \times r$. Following is the algorithm.

Algorithm:

```
Matrix-Multiplication (X, Y, Z)
for i = 1 to p do
   for j = 1 to r do
```

```
    Z[i,j] := 0
    for k = 1 to q do
       Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]
```

There are three for loops in this algorithm and one is nested in the other. Hence, the algorithm takes $O(n^3)$ time to execute.

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n, for any order of n x n the algorithm uses only 7 multiplications and 18 additions. Strassen's multiplication divides the nxn matrix into four matrices of n/2 × n/2 and reduces the complexity to $O(n^{2.807})$.

9. Explain recurrence relation for Strassen's matrix.

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$T(N) = 7T(N/2) + O(N^2)$

p1 = a(f - h) **p2 = (a + b)h**
p3 = (c + d)e **p4 = d(g - e)**
p5 = (a + d)(e + h) **p6 = (b - d)(g + h)**
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

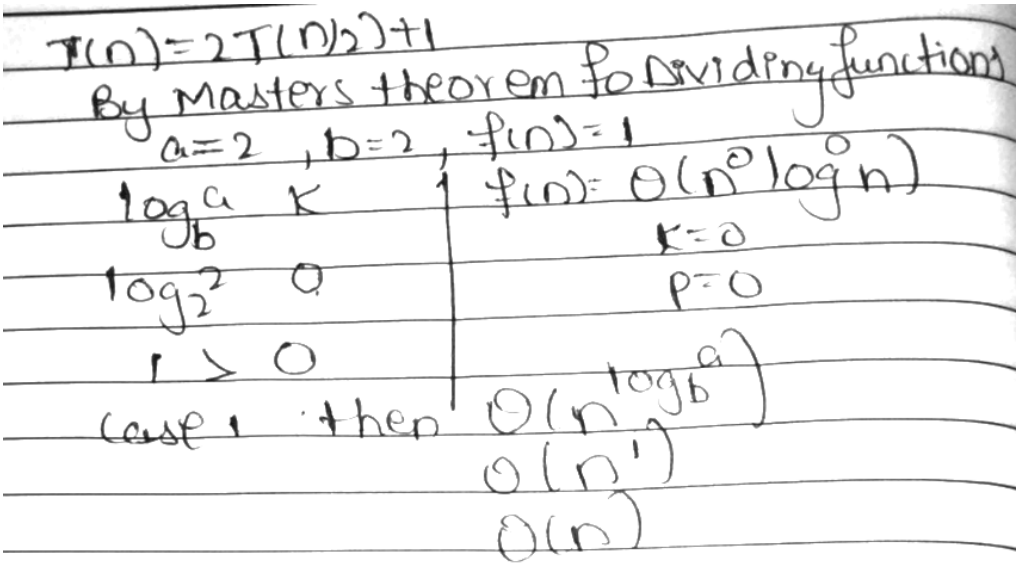
A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

From Master's Theorem, time complexity of above method is

$O(NLog7)$ which is approximately $O(N^{2.8074})$

10.Solve the following recurrence relation T (n) =2 T (n/2) + 1, and T (1) =2.



PART B

1. Define various asymptotic notations used for best case,average case and worst case analysis of algorithms.

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation: The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
  For example, for a function f(n)

O(f(n)) = { g(n) : there exists c > 0 and n_0 such that f(n) ≤ c.g(n) for all n > n_0. }

- Ω Notation: The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to

complete.

For example, for a function f(n)

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

- θ Notation: The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$

2. Explain the difference between posteriori analysis and priori analysis.

A Posteriori analysis	A priori analysis
Posteriori analysis is a relative analysis.	Piori analysis is an absolute analysis.
It is dependent on language of compiler and type of hardware.	It is independent of language of compiler and types of hardware.
It will give exact answer.	It will give approximate answer.
It doesn't use asymptotic notations to represent the time complexity of an algorithm.	It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.
The time complexity of an algorithm using a posteriori analysis differ from system to system.	The time complexity of an algorithm using a priori analysis is same for every system.

3. Explain Binary search algorithm and analyse its time complexity.

Algorithm

```
1.  Begin
2.  Set beg = 0
3.  Set end = n-1
4.  Set mid = (beg + end) / 2
5.  while ( (beg <= end) and (a[mid] ≠ item) ) do
6.  if (item < a[mid]) then
7.  Set end = mid - 1
8.  else
9.  Set beg = mid + 1
10. endif
11. Set mid = (beg + end) / 2
12. endwhile
13. if (beg > end) then
14. Set loc = -1
15. else
16. Set loc = mid
17. endif
18. End
```

Time Complexity

At **Iteration 1**,
Length of array = **n**
At **Iteration 2**,
Length of array = $n/2$
At **Iteration 3**,
Length of array = $(n/2)/2 = n/2_2$
Therefore, after **Iteration k**,
Length of array = $n/2_k$
Also, we know that after
After k iterations, the **length of array becomes 1**
Therefore
Length of array = $n/2_k$**=1**
=> **n = 2^k**

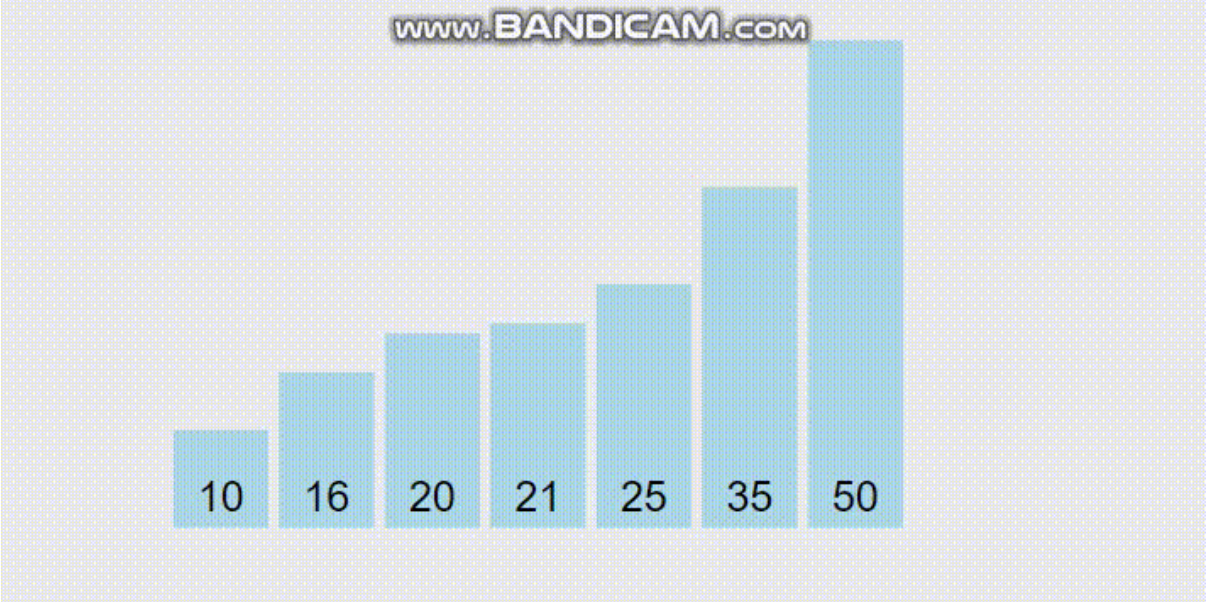
- Applying log function on both sides:
- **$\log_2$ (n) = log2 (2^k)**
- **$\log_2$ (n) = k log2 (2)**

As **($\log_a$ (a) = 1)**
Therefore,
k = $\log_2$ (n)

4. Explain quick sort algorithm and simulate it for the following data: 20, 35, 10, 16, 54, 21, 25

Algorithm

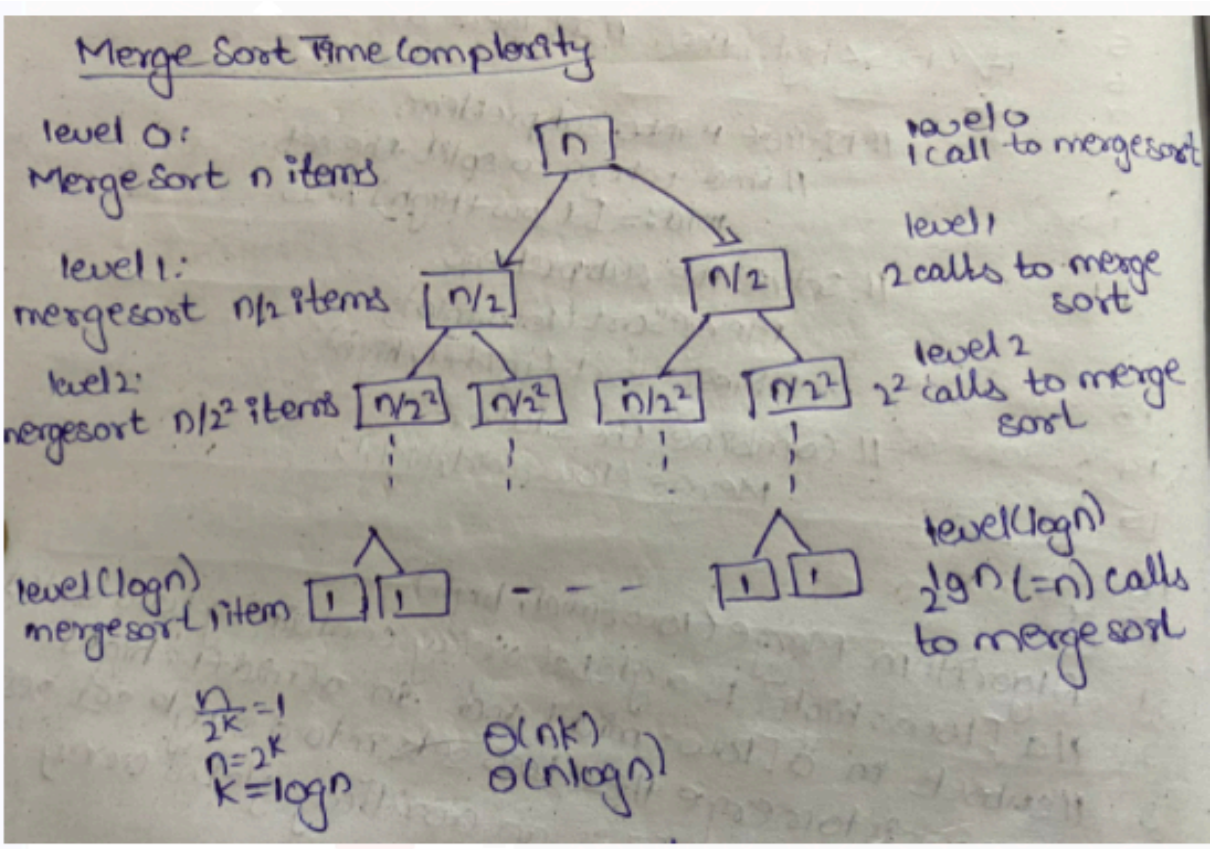
```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array, leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)
partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
 for i <- leftmostIndex + 1 to rightmostIndex
 if element[i] < pivotElement
    swap element [i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```



5. Define iterative binary search algorithm.

Same as the 3rd Question. Click here if you're viewing in Google Docs.

6. Illustrate merge sort algorithms and discuss time complexity in both worst case and average cases.



7. Explain the advantage of Strassen's matrix multiplication when compared to normal matrix multiplication for any two 16 x 16.

Refer Part A 8th Ques.

8. Explain amortised analysis and discuss how amortised complexity and actual complexity related.

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyse a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

The example data structures whose operations are analysed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

In the Hash-table, most of the time the searching time complexity is O(1), but sometimes it executes O(n) operations. When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task, but when a collision occurs, it needs O(n) times operations for collision resolution.

Classical asymptotic analysis gives worst case analysis of each operation without taking the effect of one operation on the other, whereas amortized analysis focuses on a sequence of operations, an interplay between operations, and thus yielding an analysis which is precise and depicts a micro-level analysis.

9. Define probabilistic analysis and randomised algorithms.

- In analysis of algorithms, probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or a computational problem. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm. This approach is not the same as that of probabilistic algorithms, but the two may be combined.
- A randomised algorithm is an algorithm that employs a degree of randomness as part of its logic or procedure. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random determined by the random bits; thus either the running time, or the output (or both) are random variables.

10. Organise sorted list of numbers using merge sort: 78, 32, 42, 62, 98, 12.

78	32	42	62	98	12
0	1	2	3	4	5

13. Define the Pseudo code conventions for specifying algorithms of recursive and an iterative algorithm to compute n!.

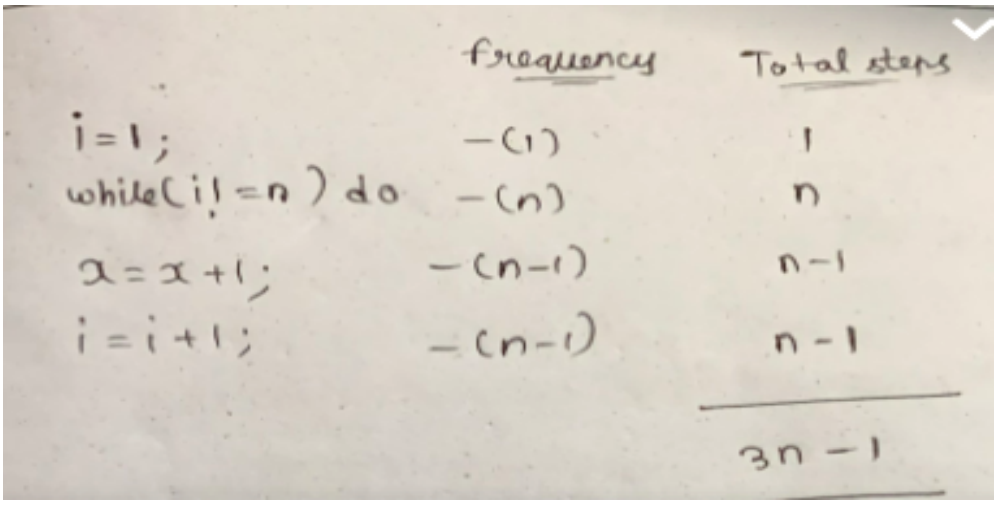
Pseudo code for a factorial number:

```
X = K
Y = 1
while X !=1 do

Y = Y * X

X = X - 1

return Y
```

By Recursion Method:

```
Fact(n)
Begin
if n == 1 then
Return ;
else
Return n*Call Fact(n-1);
end if
End
```

14. Explain the frequency counts for all statements in the following algorithm segment. i=1; while(i¡=n) do x=x+1; i=i+1.



15. What is a stable sorting method? Is merge sort a stable sorting method? Justify

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some Sorting Algorithms are stable by nature like Insertion Sort, Merge Sort and Bubble Sort etc.

Sorting algorithms are not stable like Quick Sort, Heap Sort etc.

- Merge sort is a stable algorithm but not an in-place algorithm. It requires extra array storage.
- The same element in an array maintains their original positions with respect to each other.
- Overall time complexity of Merge sort is O(nLogn).
- It is more efficient as it is in the worst case also the runtime is O(nlogn)The space complexity of Merge sort is O(n). This means that this algorithm takes a lot of space and may slower down operations for the last data sets.

16. What is the Bubble sorting method? Is bubble sort a stable sorting method? Justify

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- BubbleSort is stable.
- Stable means two equal elements are in the same order before and after the sorting.
- The list 1,1,3,2 is sorted with BubbleSort by only swapping the 3 and the 2. So the two ones are still in the same order.

17. What is Quick sort? Is quicksort the best sorting method? Justify

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

The time complexity of Quicksort is O(n log n) in the best case, O(n log n) in the average case, and O(n^2) in the worst case. But because it has the best performance in the average case for most inputs, Quicksort is generally considered the “fastest” sorting algorithm.

Quicksort is a common one for two reasons:

1) it is in-place, i.e. it does not need extra memory when sorting a huge list

2) it performs great on average.

The time complexity of Quicksort is O(n log n) in the best case, O(n log n) in the average case, and O(n^2) in the worst case. But because it has the best performance in the average case for most inputs, Quicksort is generally considered the “fastest” sorting algorithm.

18. What is the Bubble sorting method? Is bubble sort a stable sorting method? Justify

A) refer q.no.16. click here if you are using Google Docs!

19. Compare different asymptotic notations

A) refer Q.no.1 click here if you are using Google Docs!

continuation….

4. Little o notation: is used to describe an upper bound that cannot be tight. In other words, the loose upper bound of f(n).

5. Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth.

20. Differentiate time and space complexity? Justify

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

PART-C

1.Define the term algorithm

A) An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.

2.Define order of an algorithm.

A) Order of growth of an algorithm is a way of saying/predicting how execution time of a program and the space/memory occupied by it changes with the input size.

3. List asymptotic notations for big ‘Oh’, omega and theta.

A) refer 1ans part b click here if you are using Google Docs!

4.What do you mean by probability analysis?

A) refer 9 ans part b click here if you are using Google Docs!

5. Find The best case and worst case analysis for linear search.

A) Linear Search

- Best case: O(1)
- Average case: O(n)
- Worst case: O(n)

7. Define the recurrence equation for the worst case behaviour of merge sort.

A) If T(n) is the time required by merge sort for sorting an array of size n, then the .

8.Find the average case time complexity of quick sort.

A) O(n log n)

9.Define algorithm correctness.

A) A correct algorithm is one in which every valid input instance produces the correct output. The correctness must be proved mathematically.

Efficiency of an algorithm

worst case efficiency

is the *maximum* number of steps that an algorithm can take for *any* collection of data values.

Best case efficiency

is the *minimum* number of steps that an algorithm can take *any* collection of data values.

Average case efficiency

- the efficiency averaged on all possible inputs
- must assume a distribution of the input
- we normally assume uniform distribution (all keys are equally probable)

If the input has size *n*, efficiency will be a *function of n*

10.Define best case, average case and worst case efficiency of an algorithm.

A) Worst case efficiency: It is the minimum number of steps that an algorithm can take for any collection of data values.

Best case Efficiency: It is the minimum number of steps that an algorithm can take any collection of data values.

Worst case Efficiency:

- the efficiency averaged on all inputs
- must assume a distribution of the input
- we normally assume uniform distribution(**all keys are equally portable**)

11. Define the term amortised efficiency.

A) In computer science, amortised analysis is a method for analysing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortised analysis is that looking at the worst-case run time can be too pessimistic.

12. Define order of growth.

A) An order of growth is a set of functions whose asymptotic growth behaviour is considered equivalent.

13. How do you measure the runtime of an algorithm?

A) to calculate the running time, find the maximum number of nested loops that go through a significant portion of the input.

17. What is meant by divide and conquer? Give the recurrence relation for divide and conquer.

A) In divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Following are some of the examples of recurrence relations based on divide and conquer. //(doubt)

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

$$T(n) = 2T(n/2) + \sqrt{n}$$

19. Find out any two drawbacks of the binary search algorithm.

A) It employs a recursive approach which requires more stack space.

- Programming binary search algorithms is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

20. Find out the drawbacks of the Merge Sort algorithm.

A) For small datasets, merge sort is slower than other sorting algorithms.

1. For the temporary array, mergesort requires an additional space of O(n).
2. Even if the array is sorted, the merge sort goes through the entire process.

IARE '24 DISCORD TEAM

II-II

DAA MODULE 2 SOLUTIONS

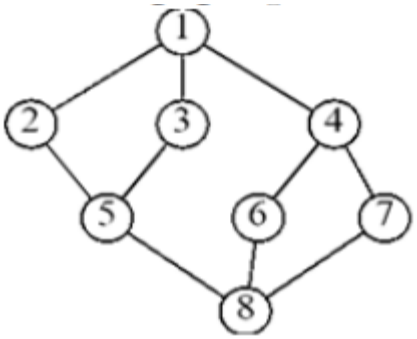
UJJWAL • ABHIRAMI • VISHAL

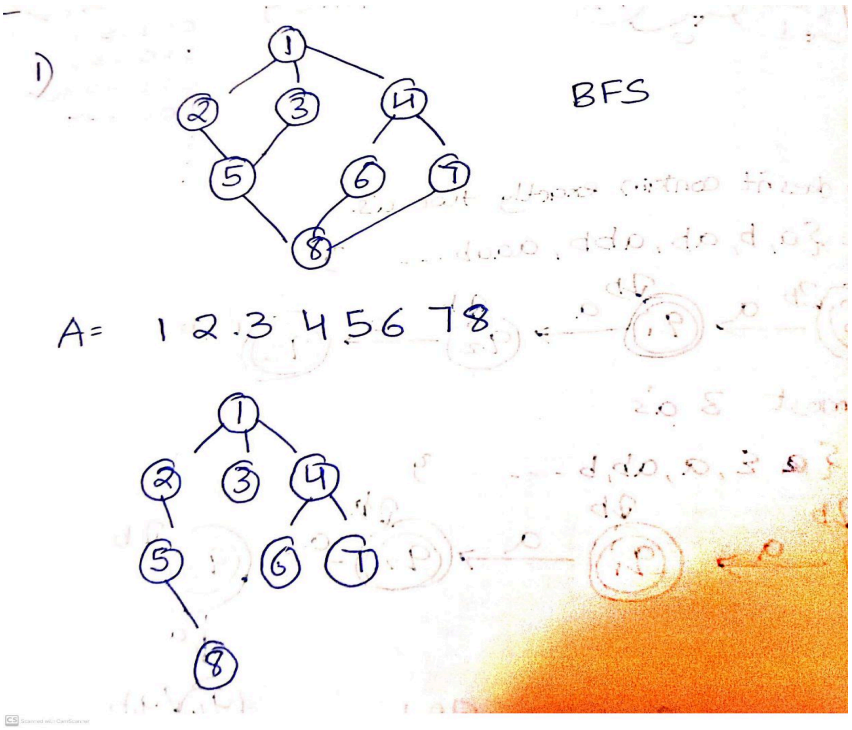
SEARCHING AND TRAVERSAL TECHNIQUES

DAA MODULE 2

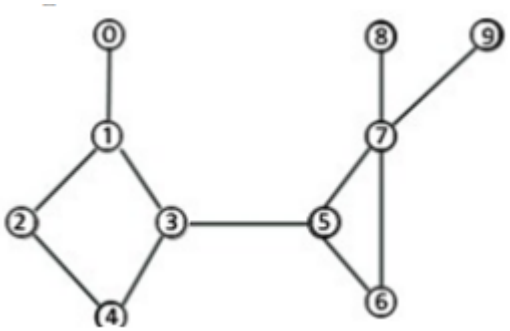
PART-A

1)Build a BFS traversal tree of the following graph.



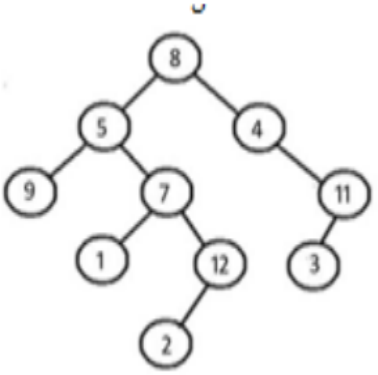


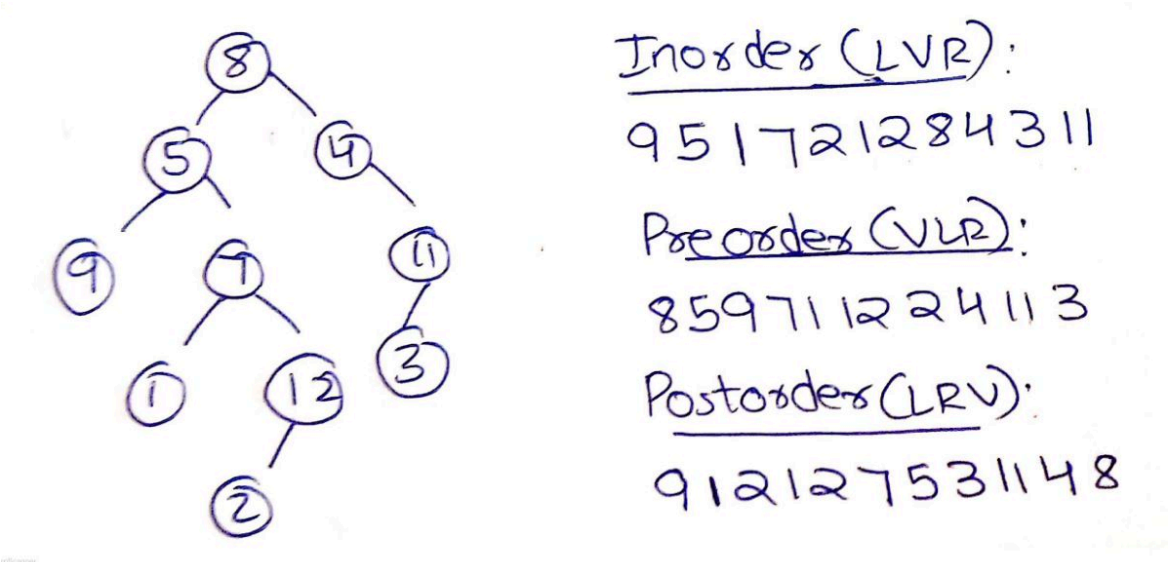
2) Identify the articulation points from the following graph



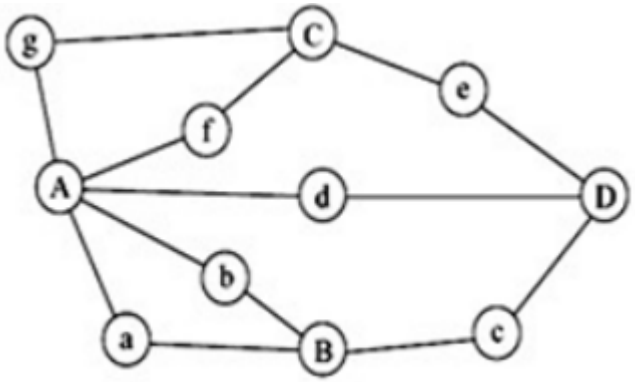
[1,3,5,7] Verified ✅

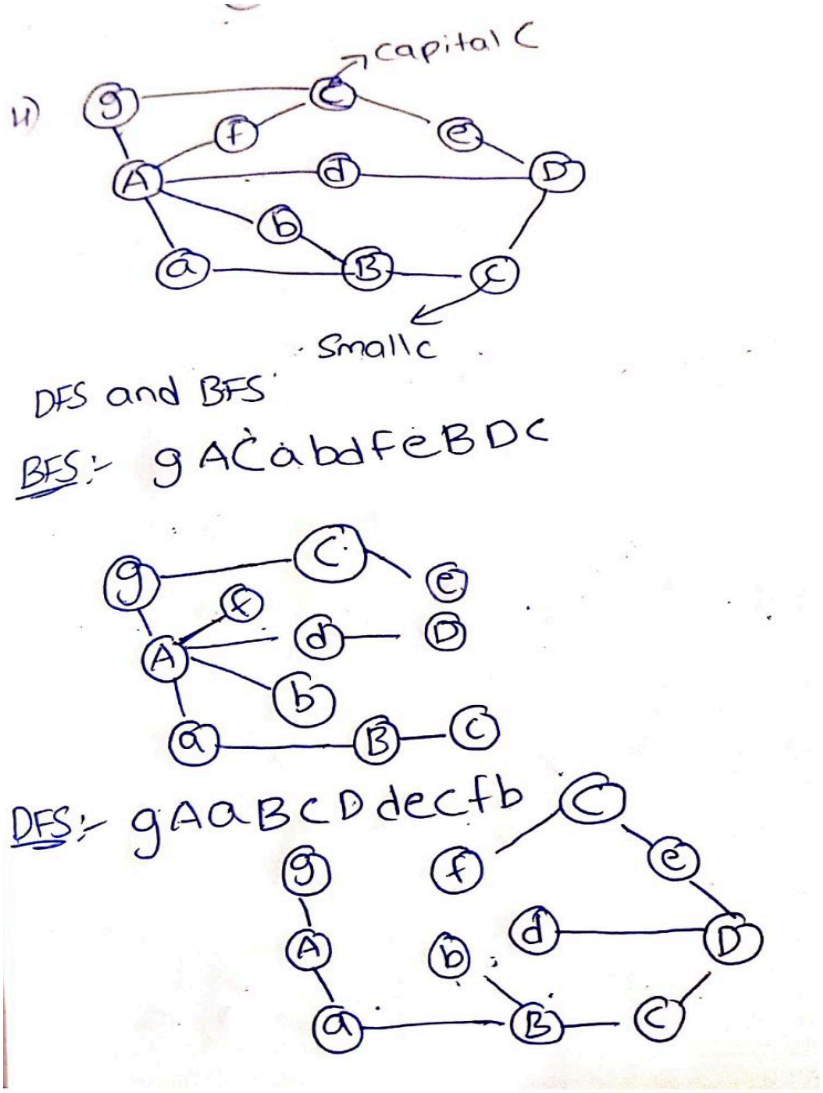
3) Organise Order, pre order, post order traversal of the following tree





4)Construct DFS and BFS traversal trees of following graph





PART-B

1)Explain the BFS algorithm with an example.

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

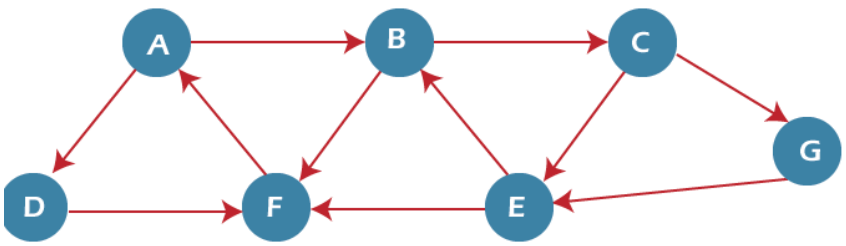
Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]
Step 6: EXIT

Example of BFS algorithm



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

Step 1 - First, add A to queue1 and NULL to queue2.

```
QUEUE1 = {A}
QUEUE2 = {NULL}
```

Step 2 -Now, delete node A from queue1 and add it into queue2. Insert all neighbours of node A to queue1.

```
QUEUE1 = {B, D}
QUEUE2 = {A}
```

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbours of node B to queue1.

```
QUEUE1 = {D, C, F}
QUEUE2 = {A, B}
```

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbours of node D to queue1. The only neighbour of Node D is F since it is already inserted, so it will not be inserted again.

```
QUEUE1 = {C, F}
QUEUE2 = {A, B, D}
```

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbours of node C to queue1.

```
QUEUE1 = {F, E, G}
QUEUE2 = {A, B, D, C}
```

Step 6 - Delete node F from queue1 and add it into queue2. Insert all neighbours of node F to queue1. Since all the neighbours of node F are already present, we will not insert them again.

```
QUEUE1 = {E, G}
QUEUE2 = {A, B, D, C, F}
```

Step 7 - Delete node E from queue1. Since all of its neighbours have already been added, we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

```
QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F, E}
```

Complexity of BFS algorithm

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of the BFS algorithm is **O(V+E)**, since in the worst case, the BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

The space complexity of BFS can be expressed as **O(V)**, where V is the number of vertices.

2) Explain depth first search algorithm with example

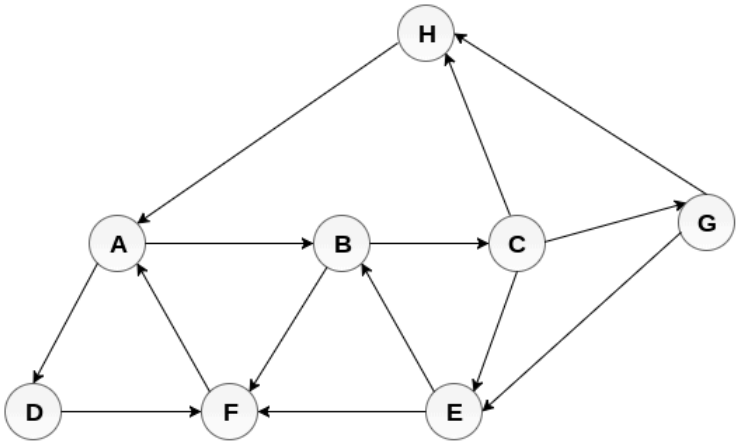
Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes deeper and deeper until we find the goal node or the node which has no children. The algorithm then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
  STATUS = 2 (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

```
STACK : H
```

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are in ready state.

```
Print H
STACK : A
```

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

```
Print A
Stack : B, D
```

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

```
Print D
Stack : B, F
```

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

```
Print F
Stack : B
```

Pop the top of the stack i.e. B and push all the neighbours

```
Print B
Stack : C
```

Pop the top of the stack i.e. C and push all the neighbours.

```
Print C
Stack : E, G
```

Pop the top of the stack i.e. G and push all its neighbours.

```
Print G
Stack : E
```

Pop the top of the stack i.e. E and push all its neighbours.

```
Print E
Stack :
```

Stack is empty now

The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

3) Explain iterative versions of binary tree traversal concept of tree traversal algorithms (inorder, algorithms and Contrast preorder and post order).

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

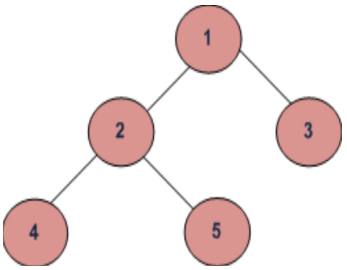
1) DFS
2) BFS

In DFS, we have:

(a) Inorder (Left, Root, Right)

(b) Preorder (Root, Left, Right)

(c) Postorder (Left, Right, Root)



Inorder Traversal:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder: In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.

Example: In order traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree. Preorder traversal for the above-given figure is 1 2 4 5 3.

Postorder Traversal (Practice):

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder : Postorder traversal is used to delete the tree.Postorder traversal is also useful to get the postfix expression of an expression tree.

Example: Postorder traversal for the above-given figure is 4 5 2 3 1.

4) Compare the approaches of BFS and DFS methods and derive the time complexity of both methods for the inputs of adjacency list and adjacency matrix separately.

BFS	DFS
A graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes	An algorithm that starts with the initial node of the graph and then goes deeper and deeper until finding the required node or the node which has no children
Stands for Breadth First Search	Stands for Depth First Search
Uses queue	Uses stack
Consumes more memory	Consumes les memory
Focuses on visiting the oldest unvisited vertices first	Focuses on visiting the vertices along the edge in the beginning

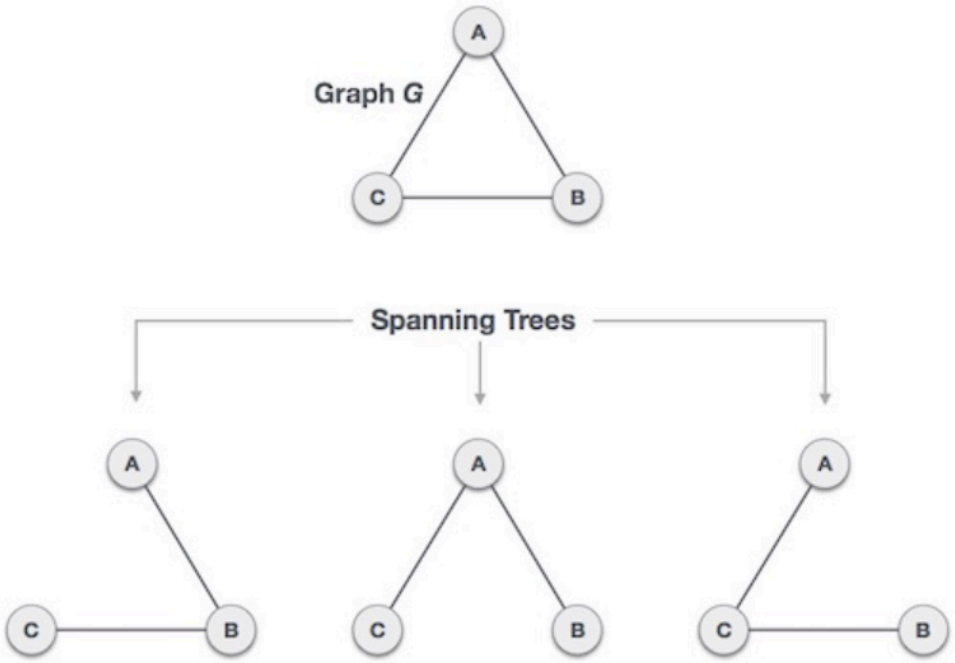
5) Explain BFS and spanning trees in detail.

BFS explanation is already in PART(B) →1

SPANNING TREE:

A spanning tree is a subset of Graph G, which has all the vertices covered with a minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



A complete undirected graph can have a maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).

- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

6) Explain weighting rule for finding UNION of sets and collapsing rule

Weighting rule for Union:

If the number of nodes in the tree with root i is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j. To implement the weighting rule we need to know how many nodes are there in every tree. To do this we maintain a "count" field in the root of every tree. If 'i' is the root then count[i] equals the number of nodes in the tree with roots 'i'. Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i j using the weighted rule
//P[i]=-count[i] andp[j]=-count[j]
{
temp:=P[i]+P[j];
if (P[i]>P[j])then {
// i has fewer nodes P[i]:=j;
```

```
P[j]:=temp;
}
else {
// j has fewer nodes P[j]:=i;
}
}
```

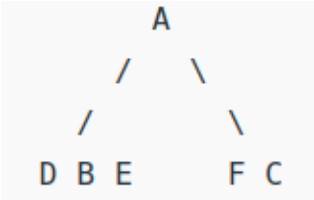
7) How to construct a binary tree from inorder and preorder traversals.

Let us consider the below traversals:

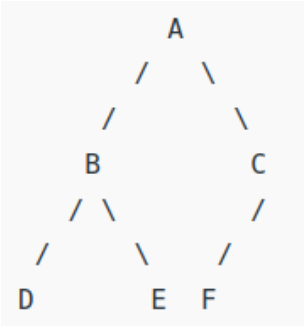
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F

In a Preorder sequence, the leftmost element is the root of the tree. So we know ‘A’ is the root for given sequences. By searching ‘A’ in the Inorder sequence, we can find out all elements on the left side of ‘A’ are in the left subtree, and elements on right in the right subtree. So we know the below structure now.



We recursively follow the above steps and get the following tree.



(not much info here, better to go check out youtube)

5.7 Construct Binary Tree from Preorder and Inorder traversal with examp...

8) Explain about DFS and spanning trees.

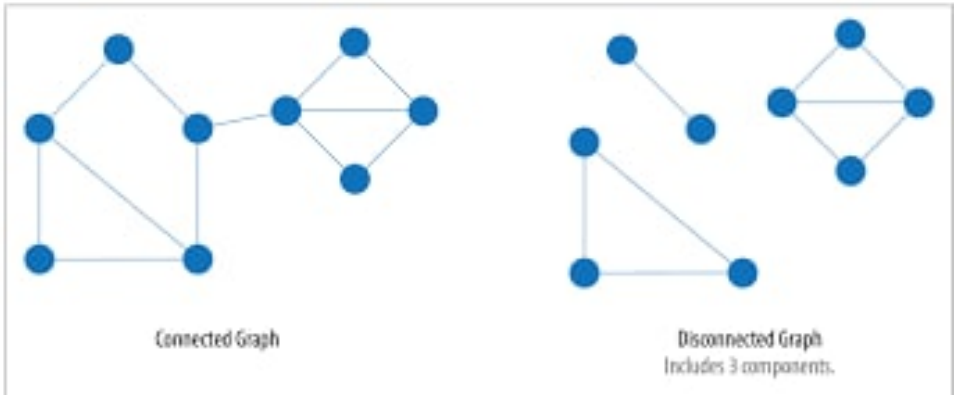
refer to PART(B) → 2 and PART(B) →5

9) Illustrate how to identify given graph is connected or not

In an undirected graph G, two vertices u and v are called connected if G contains a path from u to v. Otherwise, they are called disconnected.

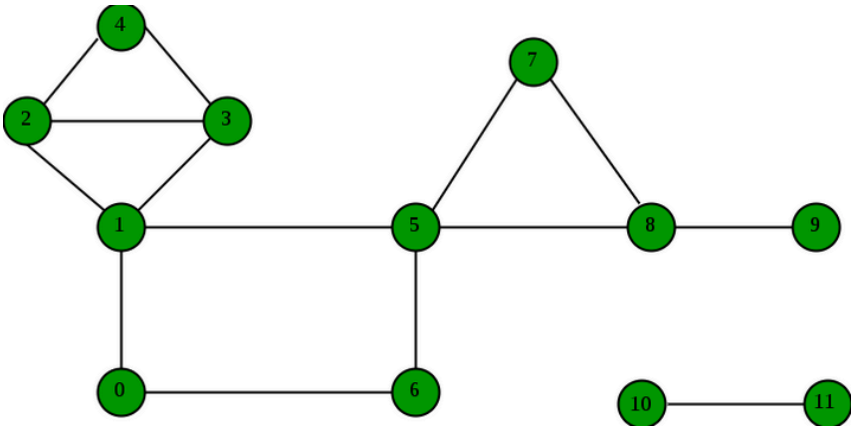
If the two vertices are additionally connected by a path of length 1, i.e. by a single edge, the vertices are called adjacent.

A graph is said to be connected if every pair of vertices in the graph is connected. This means that there is a path between every pair of vertices. An undirected graph that is not connected is called disconnected. An undirected graph G is therefore disconnected if there exist two vertices in G such that no path in G has these vertices as endpoints. A graph with just one vertex is connected. An edgeless graph with two or more vertices is disconnected.



10) Explain the concept of biconnected component with example

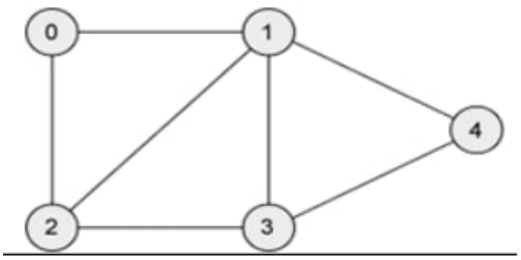
A biconnected component of a graph is a connected subgraph that cannot be broken into disconnected pieces by deleting any single node (and its incident links).



In above graph, following are the biconnected components:

- 4-2 3-4 3-1 2-3 1-2
- 8-9
- 8-5 7-8 5-7
- 6-0 5-6 1-5 0-1
- 10-11

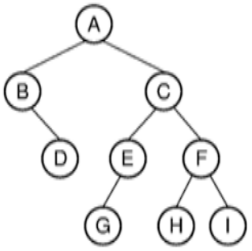
11) Develop a program to print all the nodes reachable from a given starting node in a bigraph using the BFS method.



Python Program for BFS Implementation of the above graph:

```
graph = {
  '4' : ['1','3'],
  '1' : ['0', '4', '2', '3'],
  '3' : ['2', '1', '4'],
  '2' : ['0', '1', '3'],
  '0' : ['1', '2'],
}

visited = [] #List for visited nodes.
queue = []   #Initialise a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

12) Develop a program to perform various tree traversal algorithms for a given tree.

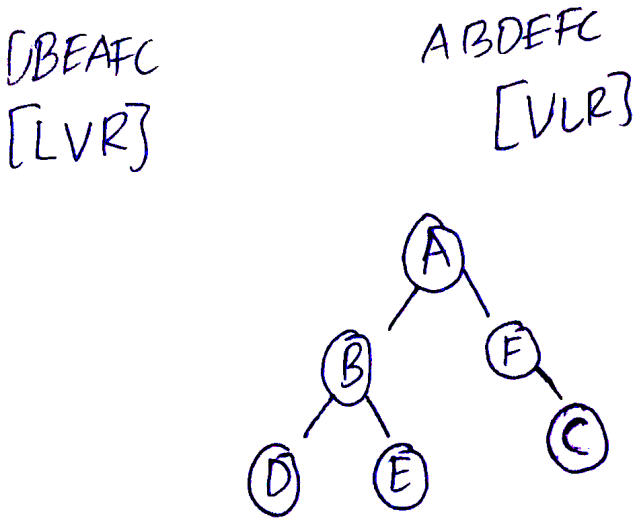


Python Code for Tree Traversal Algorithms

```
class Node:
```

```
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val)
        printInorder(root.right)

def printPostorder(root):
    if root:
        printPostorder(root.left)
        printPostorder(root.right)
        print(root.val)

def printPreorder(root):
    if root:
        print(root.val)
        printPreorder(root.left)
        printPreorder(root.right)

# Driver code
root = Node("A")
root.left = Node("B")
root.right = Node("C")
root.left.right = Node("D")
root.right.left = Node("E")
root.right.left.left = Node("G")
root.right.right = Node("F")
root.right.right.left = Node("H")
root.right.right.right = Node("I")

print("Preorder traversal of binary tree is")
printPreorder(root)

print("Inorder traversal of binary tree is")
printInorder(root)

print("Postorder traversal of binary tree is")
printPostorder(root)
```

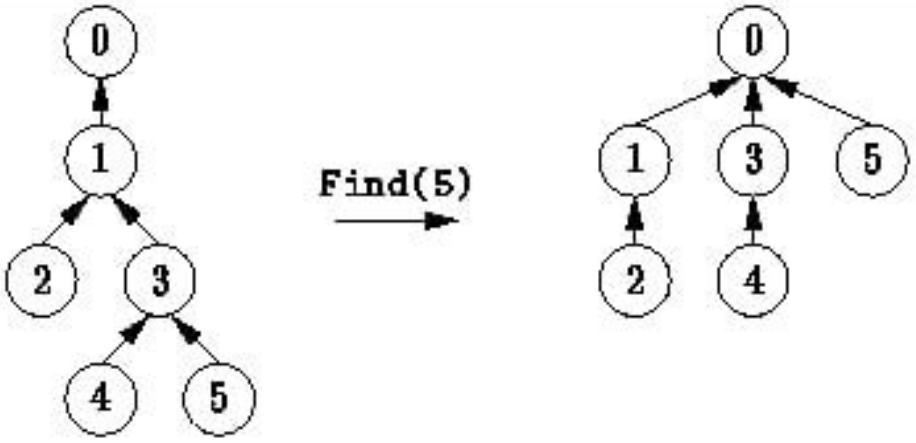
13) Construct a binary tree from the following Inorder Sequence: D B E A F C and Preorder sequence: A B D E C F.



14) Illustrate the advantage of collapse find over simple find with an example.

If, having found the root, we replace the parent pointer of the given node with a pointer to the root, the next time we do a Find it will be more efficient. In fact, we can go one step further and replace the parent pointer of every node along the search path to the root. This is called a collapsing find operation.

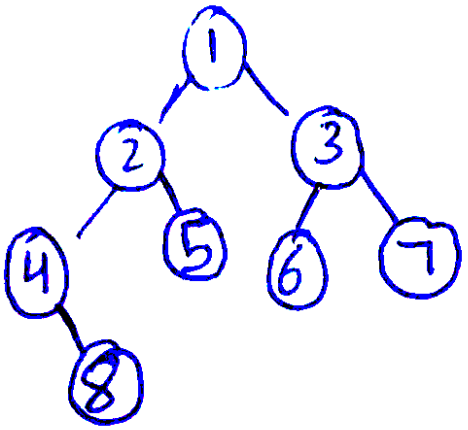
Effect of a collapsing find operation. After the find, all the nodes along the search path are attached directly to the root. I.e., they have had their depths decreased to one. As a side-effect, any node which is in the subtree of a node along the search path may have its depth decreased by the collapsing find operation. The depth of a node is never increased by the find operation. Eventually, if we do enough collapsing find operations, it is possible to obtain a tree of height one in which all the non-root nodes point directly at the root.



15) Construct a binary tree from the following Inorder sequence: 4, 8, 2, 5, 1, 6, 3, 7 and Postorder sequence: 8, 4, 5, 2, 6, 7, 3, 1.

Inorder sequence: 4, 8, 2, 5, 1, 6, 3, 7

Postorder sequence: 8, 4, 5, 2, 6, 7, 3, 1



16) Explain step count method and analyse the time complexity when two nxn matrices are added.

Step Count Method

The step count method is one of the methods to analyse the algorithm. In this method, we count the number of times one instruction is executing. From that we will try to find the complexity of the algorithm.

Given below is the step count table of general matrix addition algorithm:

Statement	s/e	Frequency	Total steps

Void add(int a[ ][MAX_SIZE] · · ·)
{
int i, j;
for (i = 0; i < row, i++)
for (j=0; j< cols; j++)
c[i][j] = a[i][j] + b[i][j];
}
	0
	0
	0
	1
	1
	1
	0
		0
		0
		0
		rows+1
		rows · (cols+1)
		rows · cols
		0
			0
			0
			0
			rows+1
			rows · cols+rows
			rows · cols
			0

Total			2rows · cols+2rows+1

Here, the number of rows and columns are equal to n. Therefore, the total time complexity for matrix addition is as follows:

Time Complexity of nxn Matrix Addition: $\mathbf{2n^2 + 2n + 1}$

Time Complexity: $\mathbf{n^2}$

18) What is meant by divide and conquer? Give the recurrence relation for divide and conquer.

The divide-and-conquer technique involves taking a large-scale problem and dividing it into similar sub-problems of a smaller scale, and recursively solving each of these sub-problems. Generally, a problem is divided into sub-problems repeatedly until the resulting sub-problems are very easy to solve.
This type of algorithm is so called because it divides a problem into several levels of sub-problems, and conquers the problem by combining the solutions at the various levels to form the overall solution to the problem. The control abstraction for divide and conquer technique is DANDC(P),where P is the problem to be solved.

```
DANDC (P)  {
if SMALL (P) then return S (p);
else {
divide p into smaller instances p1, p2, .... Pk, k>=1;
apply DANDC to each of these sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),...., DANDC (pk));  }
}
```

If the sizes of the two subproblems are approximately equal then the computing time of DANDC is given by the recurrence relation:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

19) Explain Control abstraction of divide and conquer.

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)  {
if SMALL (P) then return S (p);
else {
divide p into smaller instances p1, p2, .... Pk, k>=1;
apply DANDC to each of these sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),...., DANDC (pk));  }
}
```

If the sizes of the two subproblems are approximately equal then the computing time of DANDC is given by the recurrence relation:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

20) Find out any two drawbacks of the binary search algorithm.

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.
The drawbacks of Binary Search algorithm are as follows:

- It employs a recursive approach which requires more stack space.
- Programming binary search algorithms is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor. (because of its random access nature)

21) Find out the drawbacks of the Merge Sort algorithm.

Merge Sort is based on the divide-and-conquer strategy. Merge sort continuously cuts down a list into multiple sublists until each has only one item, then merges those sublists into a sorted list.
The drawbacks of Merge Sort Algorithm are as follows:

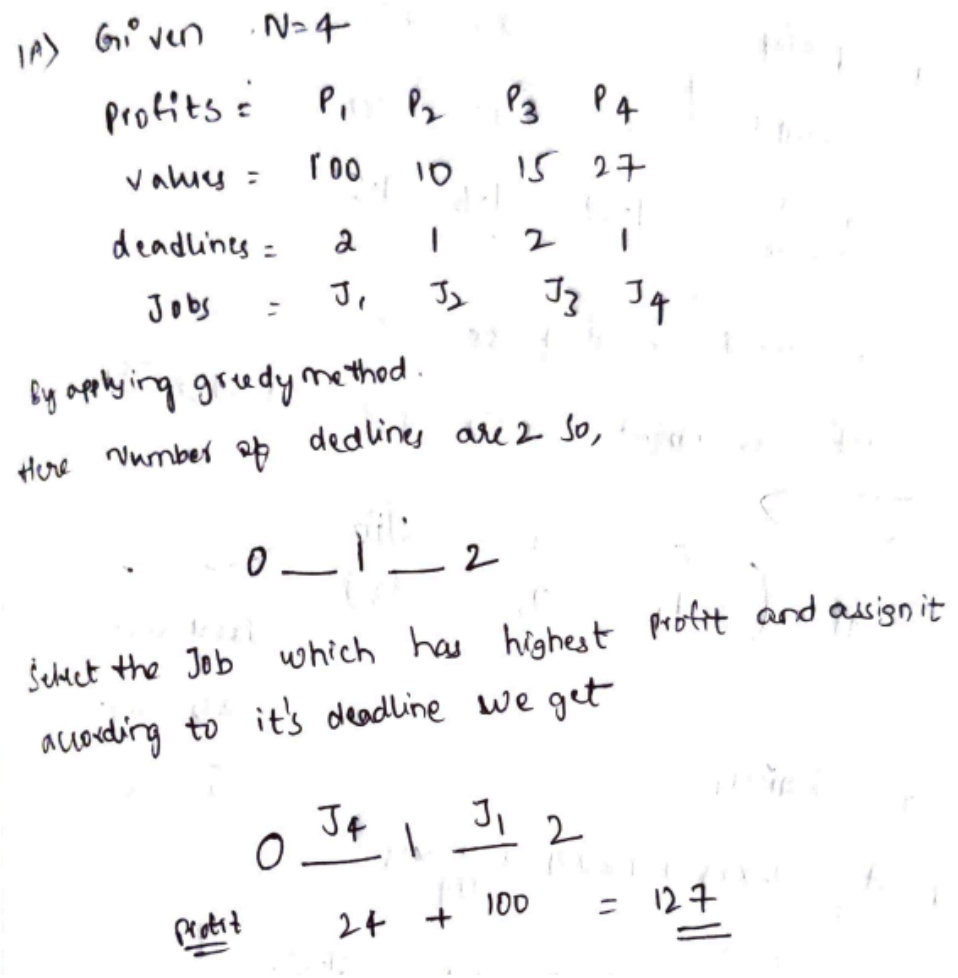
- For small datasets, merge sort is slower than other sorting algorithms.
- For the temporary array, mergesort requires an additional space of O(n).
- Even if the array is sorted, the merge sort goes through the entire process.

Many Pending

DAA MODULE 3

PART A (CIE 2)

1. Identify the optimal solution for job sequencing with deadlines using greedy methods. N=4, profits (p1,p2 ,p3,p4) = (100,10, 15,27), Deadlines (d1,d2, d3,d4) = (2,1,2,1)



2. Identify the optimal solution for knapsack problem using greedy method N=3, M= 20, (p1,p2,p3)= (25,24,15), (w1,w2,w3) =(18,15,10)

2A] Given N=3, M=20

Object	1	2	3
(Pi) Profits	25	24	15
Weights	18	15	10
Profit / Weight	1.38	1.6	1.5

Given total weight of knapsack (m) = 20

Select the object which has highest p/w value.

xi (0 1 5/10)
(x_1 x_2 x_3)

Totalweight – wei

20 – 15 = 5

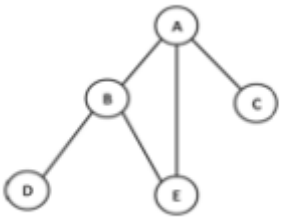
5 – 5 = 0

$p = \sum x_i \times p_i$

profit = $0 \times 25 + 1 \times 24 + \frac{5}{10} \times 15$

$= 0 + 24 + 7.5 = 31.5$.

6. Identify whether a given graph is connected or not using the DFS method.



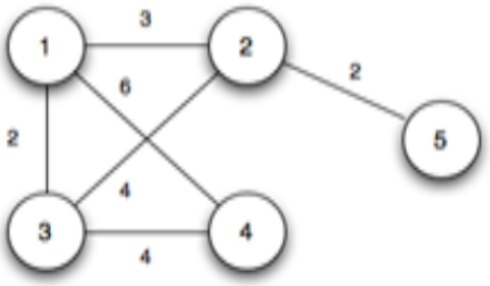
Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. This process continues until a vertex with no adjacent unvisited vertices is encountered.

Thus, if all the vertices of the above graph are connected, then, DFS must cover all the vertices of the graph.

The DFS of the above tree is: A → B → D → E → C.

Thus, all the vertices are covered and so, the graph is connected.

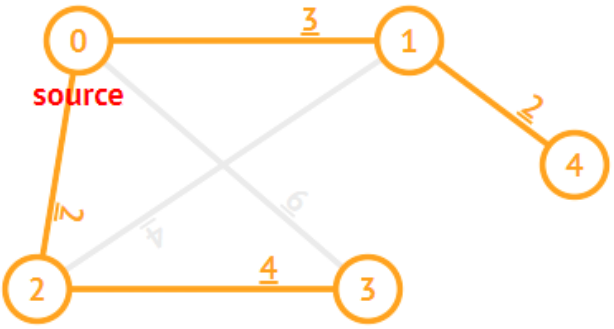
7. Construct Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.



Prim's algorithm is a greedy algorithm that starts from one vertex and continues to add the edges with the smallest weight until the goal is reached.

The steps to implement the prim's algorithm are given as follows -

- First, we have to initialise an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.



8. Apply Optimal binary search tree algorithm and compute wij, cij, rij, 0¡=i¡=j¡=4,p1=1/10, p2=1/5, p3=1/10, p4=1/120, q0=1/5, q1=1/10, q2=1/5, q3=1/20,q4=1/20.

9. Construct optimal binary search tree for (a1, a2, a3, a4) = (do, if,int, while), p(1 : 4) = (3,3,1,1) q(0 : 4)= (2,3,1,1,1)

10.Solve the solution for 0/1 knapsack problem using dynamic programming (p1,p2,p3, p4) = (11, 21, 31, 33), (w1, w2, w3, w4) = (2, 11, 22, 15), M=40, n=4

PART B (CIE 2)

8. Demonstrate Bellman Ford algorithm to compute shortest path.

The problem of Dijkstra's algorithm is that it cannot deal with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suits well for distributed systems. But the time complexity of Bellman-Ford is O(VE), which is more than Dijkstra's greedy algorithm with a time complexity of O((V + E)LogV) with the use of Fibonacci Heap.

<PENDING>

9. Explain the greedy method for generating shortest paths.

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Take a simple example and find shortest path using Dijkstra's Algorithm

10. Explain the time complexities of Prim's and Kruskal's algorithms

Prim's algorithm is popular as a greedy algorithm that helps in discovering the smallest spanning tree for a weighted undirected graph. That means, this algorithm tends to search the subgroup of the edges that can construct a tree, and the complete poundage of all the edges in the tree should be minimal.

> The time complexity of Prim's algorithm is $O(V^2)$.

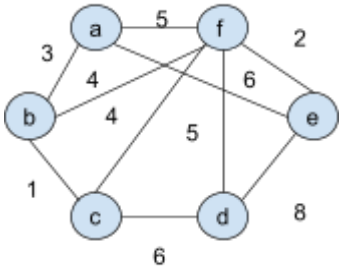
Kruskal's Algorithm is used to discover the smallest spanning tree of a connected graph and the spanning forest of an undirected edge-weighted graph. Basically, it accepts a graph as input and discovers the subgroup of the edges of that graph.

> The time complexity of Kruskal's algorithm is O(E log V).

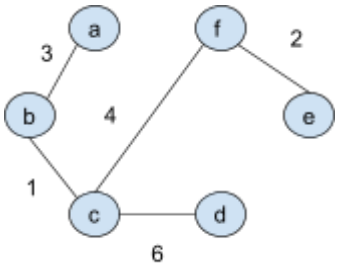
11. Compare Prim's and Kruskal's Algorithms.

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.
It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.
Prim's algorithm prefer list data structures.	Kruskal's algorithm prefer heap data structures.

12. Make use of Prim's algorithm to find minimum cost spanning tree for a graph G(6,10) with vertices named as a,b,c,d,e,f and edges ab=3,bc=1, af=5,ae=6,ed=8, fe=2,fd=5, cd=6,cf=4 and bf=4 by showing results in each stages.



Minimum shortest path



The minimum shortest path : 3+1+4+2+6=16

13. Make use of the control abstraction for the subset paradigm using greedy methods. Solve the job sequencing with the deadline problem using a greedy method for the given data N=7,P=3,5,20,18,1,6,30 are profits and D=1,3,4,3,5,1,2 are deadlines respectively.

Solution: Given

	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

Sort the jobs as per the decreasing order of profit

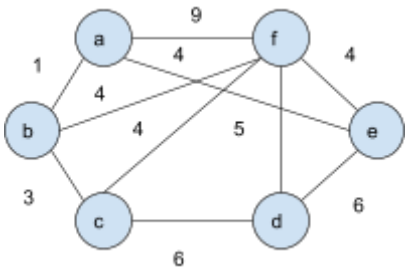
	J_7	J_3	J_4	J_6	J_2	J_1	J_5
Profit	30	20	18	6	5	3	1
Deadline	2	4	3	1	3	1	2

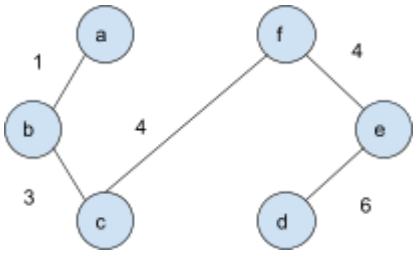
Maximum deadline is 4. Therefore create 4 slots. Now allocate jobs to highest slot, starting from the job of highest profit

Select Job 7 – Allocate to slot-2
Select Job 3 – Allocate to slot-4
Select Job 4 – Allocate to slot-3
Select Job 6 – Allocate to slot-1 Total profit earned is = 30+20+18+6=74

Slot	1	2	3	4
Job	J_6	J_7	J_4	J_3

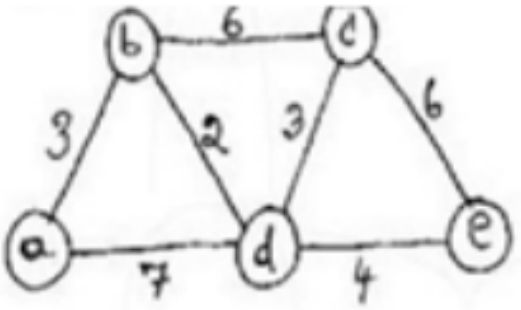
14. Identify minimum cost spanning tree for a graph G(6,10)with vertices named as a,b,c,d,e,f and edges ab=1,bc=3, af=9,ae=4,ed=6, fe=4,fd=5,cd=6,cf=4 and bf=4 using Kruskal's algorithm and showing results in each stages.



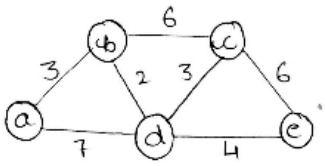


The minimum cost spanning tree is : 1+3+4+4+6=18

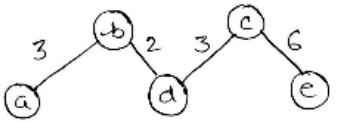
15. Identify the shortest path from source a to all other vertices in the graph shown in below Fig. Using greedy methods. Give the greedy criterion used.



15) Using Djikstra's Algorithm,



visited	a	b	c	d	e
{a}	0	3	∞	7	∞
{a, b}	0	3	9	5	∞
{a, b, d}	0	3	8	5	9
{a, b, d, c}	0	3	8	5	14
{a, b, d, c, e}	0	3	8	5	14



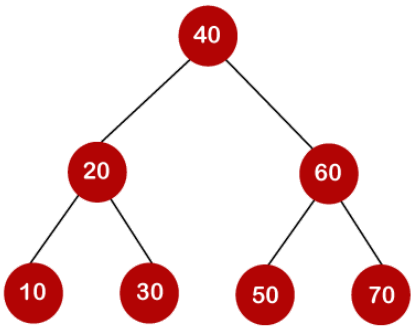
Shortest path = 14

16. Explain optimal binary search tree algorithm with example

Optimal Binary Search Tree extends the concept of Binary search tree. Binary Search Tree (BST) is a nonlinear data structure which is used in many scientific applications for reducing the search time. In BST, the left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.

Optimal Binary Search Tree (OBST) is very useful in dictionary search. The probability of searching is different for different words. OBST has great application in translation. If we translate the book from English to German, equivalent words are searched from the English to German dictionary and replaced in translation. Words are searched the same as in binary search tree order.

Lets understand through example



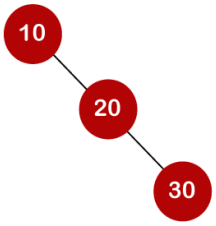
In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

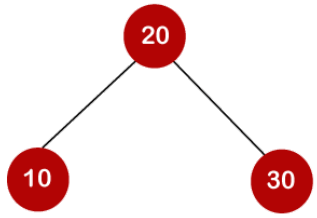
When we use the above formula, then it is found that a total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



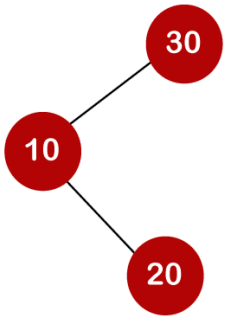
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



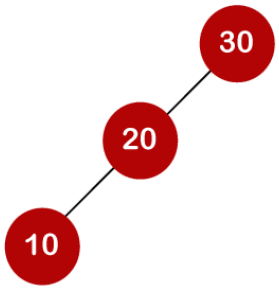
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

for more info go to Optimal Binary Search Tree - javatpoint

17. Explain 0/1 knapsack problem with an example.

Knapsack problem is also called the rucksack problem.

It is a problem in combinatorial optimization.

Knapsack problem states that: Given a set of items, each with a mass and a value, determine the number 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.

There are two versions of the problem:

a. 0/1 Knapsack Problem: Items are indivisible; you either take an item or not. Some special instances can be solved with dynamic programming.

b. Fractional knapsack problem: Items are divisible; you can take any fraction of an item.

0/1 Knapsack Problem:

i. In the 0/1 Knapsack problem, items can be entirely accepted or rejected.

ii. Given a knapsack with maximum capacity W, and a set S consisting of n items.

iii. Each item has some weight wiand benefit value bi(all wiand W are integer values).

iv. The problem is how to pack the knapsack to achieve the maximum total value of packed items.

v. For solving the knapsack problem we can generate the sequence of decisions in order to obtain the optimum selection.

vi. Let Xn be the optimum sequence and there are two instances {Xn} and {Xn-1, Xn-2... X1}.

vii. So from {Xn-1, Xn-2... X1} we will choose the optimum sequence with respect to Xn.

viii. The remaining set should fulfil the condition of filling Knapsack of capacity W with maximum profit.

ix. Thus, 0/1 Knapsack problem is solved using the principle of optimality

Example:

Find the optimal solution for the 0/1 knapsack problem making use of a dynamic programming approach. Consider-

n = 4, w = 5 kg, (w1, w2, w3, w4) = (2, 3, 4, 5), (p1, p2, p3, p4) = (3, 4, 5, 6)

Answer:

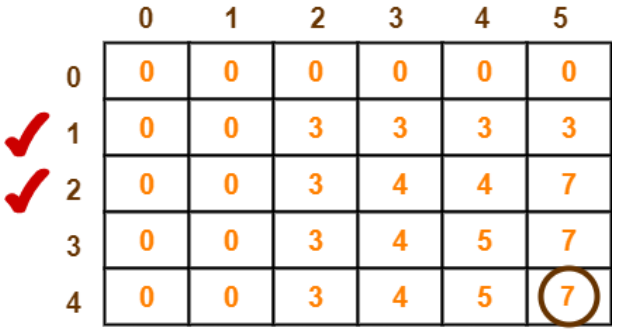
- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0th row and 0th column with 0.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Start filling the table row wise top to bottom from left to right using the formula-

$$\mathbf{T(i, j) = \max \{ T(i-1, j), profit_i + T(i-1, j - weight_i) \}}$$

After all the entries are computed and filled in the table, we get the following table-



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

The last entry represents the maximum possible value that can be put into the knapsack.

So, maximum possible value that can be put into the knapsack = 7.

18. Explain all pairs shortest path problem with example

The all pair shortest path algorithm, also known as Floyd-Warshall algorithm, is used to find all pair shortest path problems from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

At first the output matrix is the same as the given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is $O(V^3)$, here V is the number of vertices in the graph.

19. Explain the travelling salesman problem and discuss how to solve it using dynamic programming?

Travelling salesman problem is stated as, “Given a set of n cities and distance between each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at the starting city.”

It is not difficult to show that this problem is an NP-complete problem. There exists n! paths, a search of the optimal path becomes very slow when n is considerably large.

Each edge (u, v) in the TSP graph is assigned some non-negative weight, which represents the distance between city u and v. This problem can be solved by finding the Hamiltonian cycle of the graph.

The distance between cities is best described by the weighted graph, where edge (u, v) indicates the path from city u to v and w(u, v) represents the distance between cities u and v.

Let us formulate the solution of TSP using dynamic programming.

Algorithm for Travelling salesman problem

Step 1:

Let d[i, j] indicate the distance between cities i and j. Function C[x, V – { x }]is the cost of the path starting from city x. V is the set of cities/vertices in a given graph. The aim of TSP is to minimise the cost function.

Step 2:

Assume that graph contains n vertices V1, V2, ..., Vn. TSP finds a path covering all vertices exactly once, and at the same time it tries to minimise the overall travelling distance.

Step 3:

Mathematical formula to find minimum distance is stated below:

$C(i, V) = \min \{ d[i, j] + C(j, V – \{ j \}) \}, j \in V$ and $i \notin V$.

The TSP problem possesses the principle of optimality, i.e. for d[V1, Vn] to be minimum, any intermediate path (Vi, Vj) must be minimum.

From following figure, d[i, j] = min(d[i, j], d[i, k] + d[k, j])

Dynamic programming always selects the path which is minimum.

20. Explain matrix chain multiplication with examples.

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

IARE '24 DISCORD TEAM

II-II

DAA MODULE 4 SOLUTIONS

VISHAL • UJJWAL

BACKTRACKING AND BRANCH AND BOUND

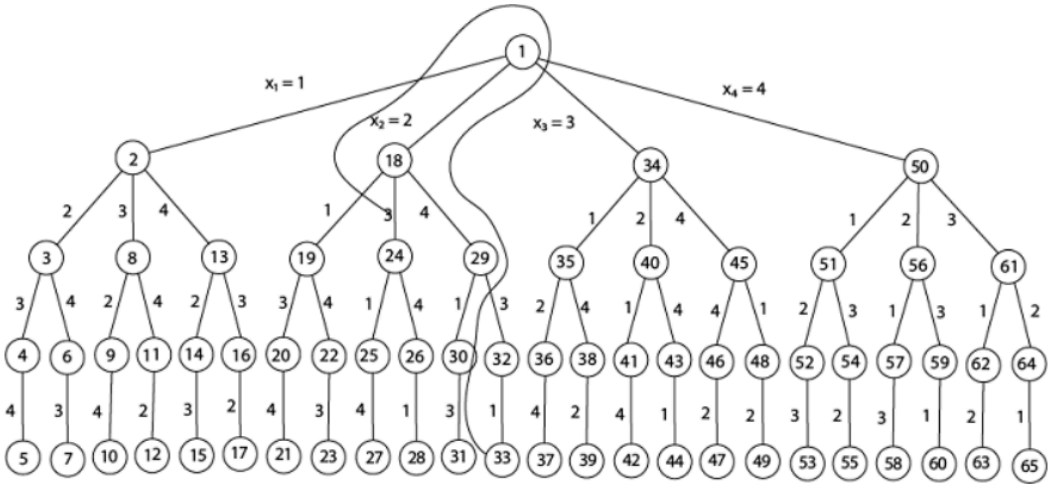
Few Pending

DAA MODULE 4

PART A

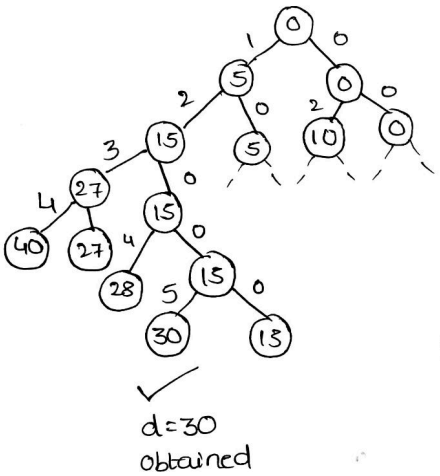
1) Build the state space tree generated by the 4 queen's problem.

Answer is (2,4,1,3) or its mirror image (3,1,4,2)



2) Apply the backtracking algorithm to solve the following instance of the sum of subsets problem S=5,10,12,13,15,18 and d=30

2) S = 5, 10, 12, 13, 15, 18 ; d = 30



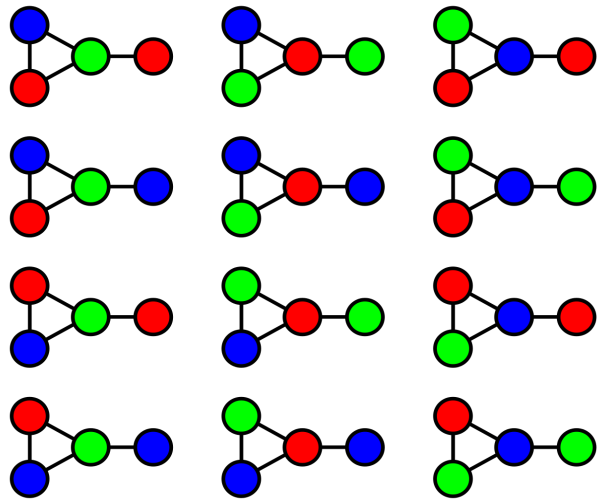
Subsets = {5, 10, 15}

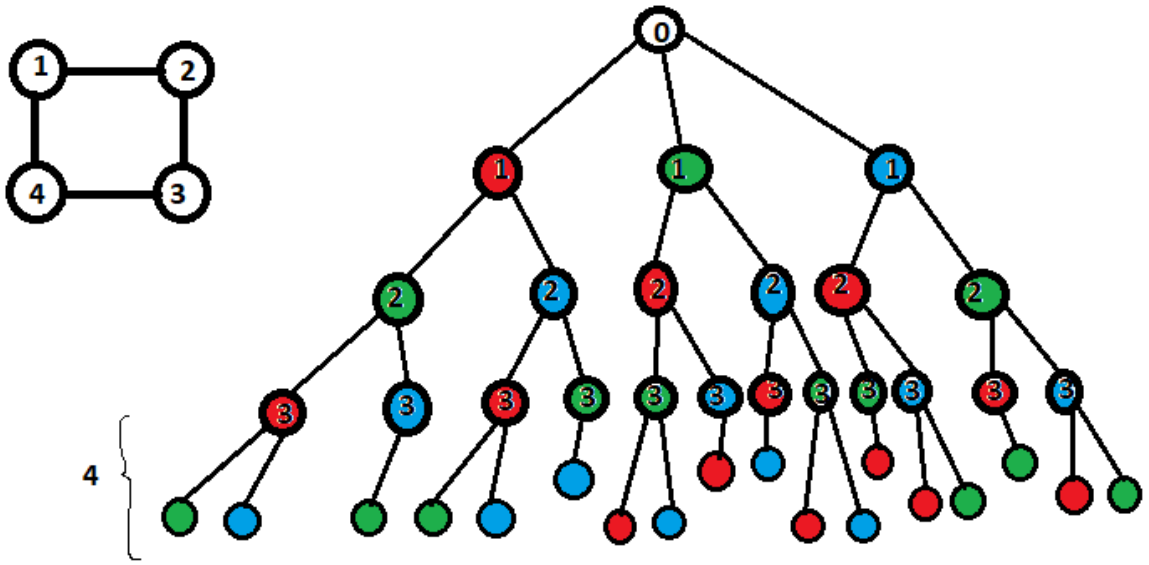
Order = {1, 1, 0, 0, 1, 0}

Sum of subsets = 30

d=30 obtained

3) Build the state space tree and generate all possible 3-color,4-node graph.





4) Question incomplete

5) Solve the following instance of a travelling salesperson problem using Least Cost Branch and Bound.

Wrong values taken for Matrix from QB. The Procedure is correct. Please follow accordingly.

5)

$$\begin{bmatrix} \infty & 12 & 5 & 7 \\ 11 & \infty & 3 & 6 \\ 4 & 9 & \infty & 18 \\ 10 & 3 & 2 & \infty \end{bmatrix} \begin{matrix} 5 \\ 3 \\ 4 \\ 2 \end{matrix} \Rightarrow \begin{bmatrix} \infty & 7 & 0 & 2 \\ 8 & \infty & 0 & 3 \\ 0 & 5 & \infty & 14 \\ 8 & 1 & 0 & \infty \end{bmatrix}$$

14

$0 \quad 1 \quad 0 \quad 2 = 3$

Reduced Cost

$\Rightarrow 14 + 3 = 17$

Reduced Matrix :

$$\begin{matrix} & 1 & 2 & 3 & 4 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} \infty & 6 & 0 & 0 \\ 8 & \infty & 0 & 1 \\ 0 & 4 & \infty & 12 \\ 8 & 0 & 0 & \infty \end{bmatrix}$$

Cost = 17

$1 \longrightarrow 2$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 \\ 0 & \infty & \infty & 12 \\ 8 & \infty & 0 & \infty \end{bmatrix} \Rightarrow \text{Cost}$$

$\Rightarrow c(1,2) + r + \hat{r}$

$\Rightarrow 6 + 17 + 1$

$\Rightarrow 24$

$0 \quad 0 \quad 0 \quad 1$

$$\longrightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ 0 & \infty & \infty & 11 \\ 8 & \infty & 0 & \infty \end{bmatrix}$$

$1 \longrightarrow 3$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & \infty & 1 \\ \infty & 4 & \infty & 12 \\ 8 & 0 & \infty & \infty \end{bmatrix} \Rightarrow \text{Cost}$$

$\Rightarrow c(1,3) + r + \hat{r}$

$\Rightarrow 0 + 17 + 9$

$\Rightarrow 26$

$8 \quad 0 \quad 0 \quad 1$

$$\longrightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 4 & \infty & 11 \\ 0 & 0 & \infty & \infty \end{bmatrix}$$

$1 \longrightarrow 4$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix} \Rightarrow \text{Cost}$$

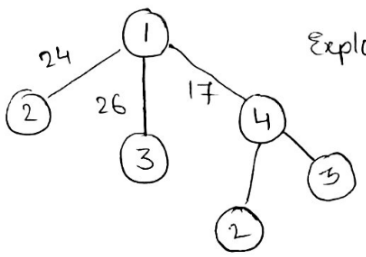
$\Rightarrow c(1,4) + r + \hat{r}$

$\Rightarrow 0 + 17 + 0$

$\Rightarrow 17$

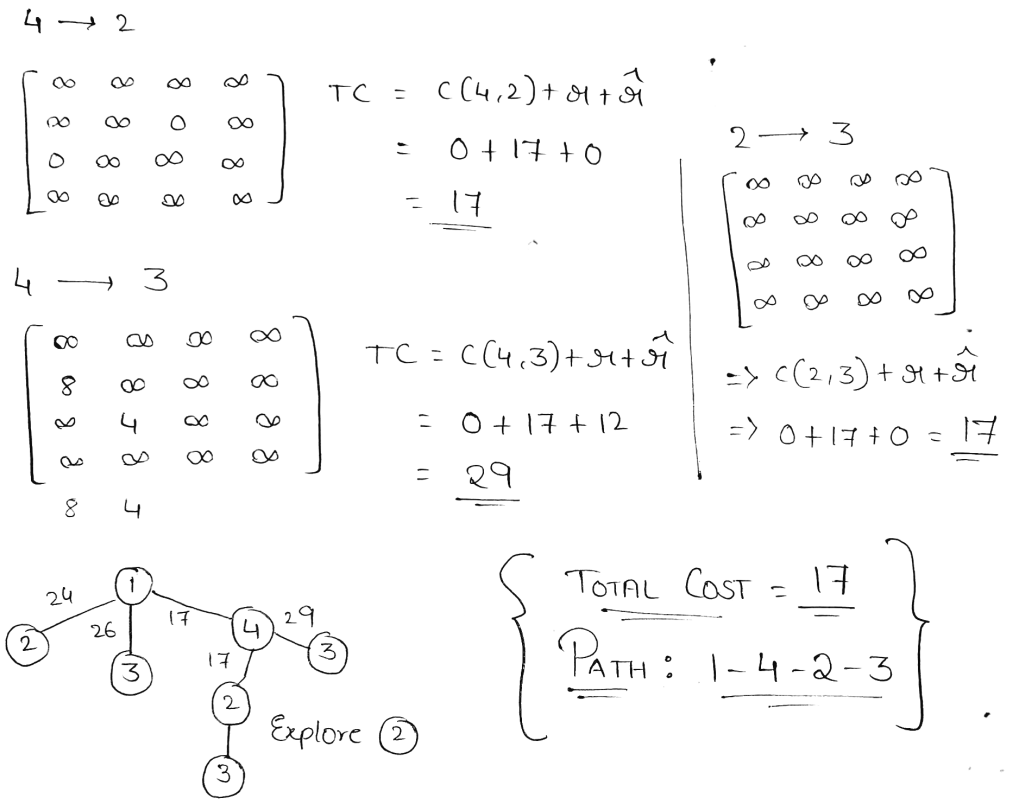
$0 \quad 0 \quad 0 \quad 0$

$\longrightarrow$ Same Matrix



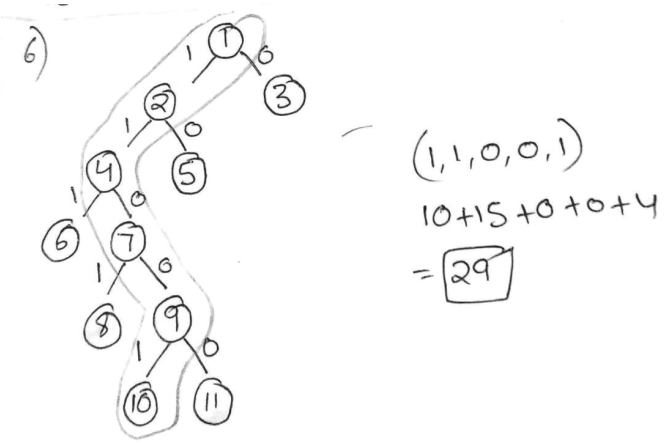
Explore (4) now

Reduced Matrix $= \begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix}$



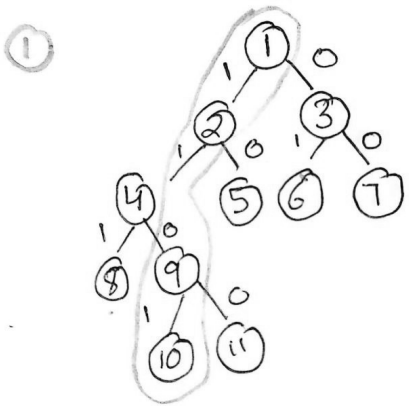
Answer with correct matrix values is 64

6) Build the state space tree generated by LCBB by the following knapsack problem n=5, (p1, p2, p3, p4, p5) = (10, 15, 6, 8,4), (w1, w2, w3, w4, w5) = (4, 6, 3, 4, 2) and m = 12.



7) Build the state space tree generated by FIFO knapsack for the instance N=4 , (P1, P2, P3, P4)= (10, 10, 12, 18) , (w1, w2,w3,w4) = (2, 4, 6, 9) , m=15

7)



(1,1,0,1)

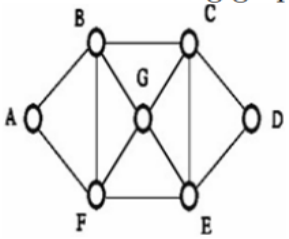
2+4+9

= 15

8) Solve the following instance of travelling salesperson problem using Least Cost Branch Bound

Same as 5th

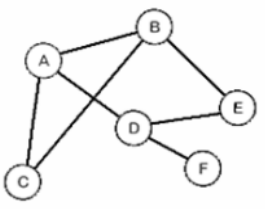
9)

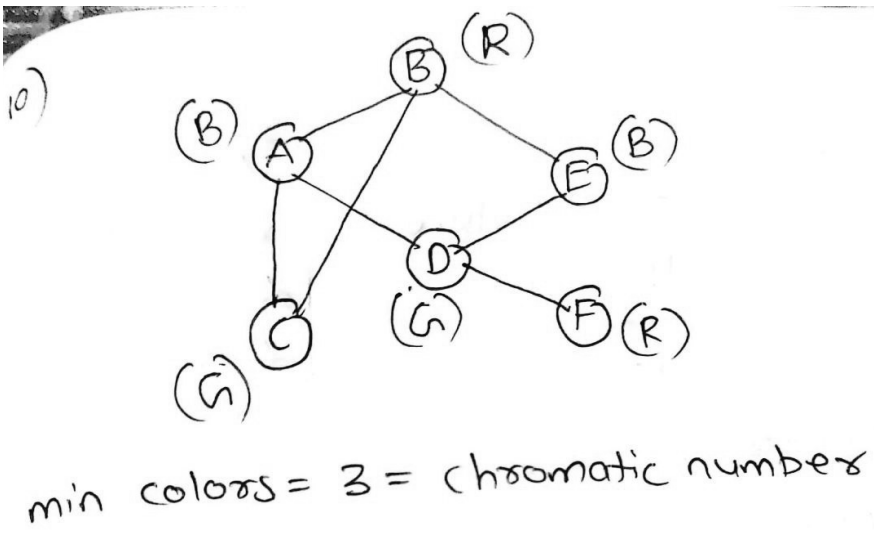
9	Identify Hamiltonian cycle from the following graph 

ABGCDEFA

etc

10)

10	Apply the backtracking algorithm to find chromatic number for the following graph 



PART B

1) Develop an algorithm for the N-queens problem using backtracking.

Refer Part B 11th q

2) Apply a backtracking method and solve subset-sum problems and discuss the possible solution strategies.

Same as Part A 2nd q

3) Apply graph colouring technique and write an algorithm for m-colouring problems.

In this problem, an undirected graph is given. There are also provided m colours. The problem is to find if it is possible to assign nodes with m different colours, such that no two adjacent vertices of the graph are of the same colours. If the solution exists, then display which colour is assigned on which vertex.

Starting from vertex 0, we will try to assign colours one by one to different nodes. But before assigning, we have to check whether the colour is safe or not. A colour is not safe whether adjacent vertices are containing the same colour.

Algorithm

isValid(vertex, colorList, col)

Input – Vertex, colorList to check, and color, which is trying to assign.

Output – True if the color assigning is valid, otherwise false.

```
Begin
   for all vertices v of the graph, do
      if there is an edge between v and i, and col = colorList[i], then
         return false
   done
   return true
End
```

4) Explain an algorithm for the Hamiltonian cycle with an example.

In an undirected graph, the Hamiltonian path is a path that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, where there is an edge from the last vertex to the first vertex.

In this problem, we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.

Algorithm

isValid(v, k)

Input – Vertex v and position k.

Output – Checks whether placing v in the position k is valid or not.

```
Begin
   if there is no edge between node(k-1) to v, then
      return false
   if v is already taken, then
      return false
   return true; //otherwise it is valid
End
```

5) Explain properties of LC search.

- Least Cost Branch and Bound is a way of finding an optimal solution from the state space tree.
- The search for an answer node can be fastened by using an "intelligent" ranking function c(.) for live nodes. The next E-node is selected on the basis of a ranking function used in LC Search.
- The node x is assigned a rank using:

  `c( x ) = f(h(x)) + g( x )`

  - where, c(x) is the cost of x.
  - h(x) is the cost of reaching x from the root and f(...) is any non-decreasing function.
  - g(x) is an estimate of the additional effort needed to reach an answer node from x.
- LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.
- An LC-search coupled with bounding functions is called an LC-branch and bound search.

6) Explain control abstraction for LC Search.

Control Abstraction for LC-search

Let **t** be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

LC search uses ĉ to find an answer node. The algorithm uses two functions

1. Least-cost()
2. Add_node().

Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

Add_node() to delete and add a live node from or to the list of live nodes.

Add_node(x)adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

7 and 8) Explain the principle of FIFO and LIFO branch and bound.

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node.
However branch and Bound differs from backtracking in two ways:

- It has a branching function, which can be a depth first search, breadth first search or based on the bounding function.
- It has a bounding function, which goes far beyond the feasibility test as a means to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node Branch and Bound is the generalisation of both graph search strategies, BFS and DFS.
A BFS-like state space search is called a FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

FIFO Branch and bound

- FIFO Branch and Bound is a BFS.
- In FIFO Branch and Bound , children of E-Node (or Live nodes) are inserted in a queue.
- Implementation of list of live nodes as a queue
  Least() Removes the head of the Queue
  Add() Adds the node to the end of the Queue

A DFS-like state space search is called a LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

LIFO Branch and bound

LIFO Branch and Bound is a D-search (or DFS).
In LIFO Branch and Bound children of E-node (live nodes) are inserted in a stack
Implementation of List of live nodes as a stack
Least() Removes the top of the stack
ADD() Adds the node to the top of the stack

9) Apply the method of reduction to solve travelling salesperson problems using branches and bonds.

Refer this link: Travelling Salesman Problem | Branch & Bound | Gate Vidyalay

10) Explain TSP using branch and bound method with example

Same as the 9th question.

11) Explain the basic principle of Backtracking and list the applications of Backtracking.

Backtracking is a technique based on algorithms to solve problems. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that don't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems:

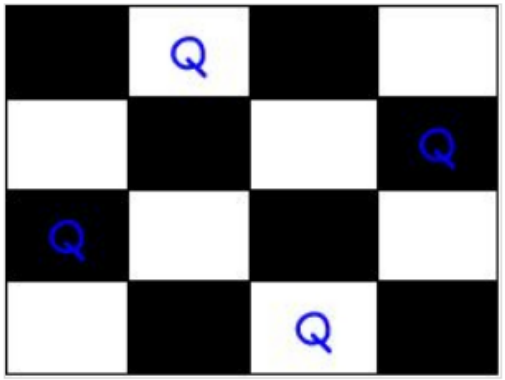
- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In a backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example:
Let's use this backtracking problem to find the solution to the N-Queen Problem.

In the N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do the 4-Queen problem.

Here, the solution is –



To solve the queen's problem, we will try placing the queen into different positions in one row. And checks if it clashes with other queens. Current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to the previous location of the queen and change its positions. And check the clash of the queens again.

Algorithm

```
Step 1 – Start from 1st position in the array.

Step 2 – Place queens in the board and check. Do,
   Step 2.1 – After placing the queen, mark the position as a part of the solution
and then recursively check if this will lead to a solution.
   Step 2.2 – Now, if placing the queen doesn't lead to a solution and trackback
and go to step (a) and place queens to other rows.
   Step 2.3 – If placing queen returns a lead to solution return TRUE.
Step 3 – If all queens are placed return TRUE.
Step 4 – If all rows are tried and no solution is found, return FALSE.
```

12) Explain Backtracking technique and solve the following instance for the subset problem s=(1,3,4,5) and d=11.

Backtracking explained in the previous question.

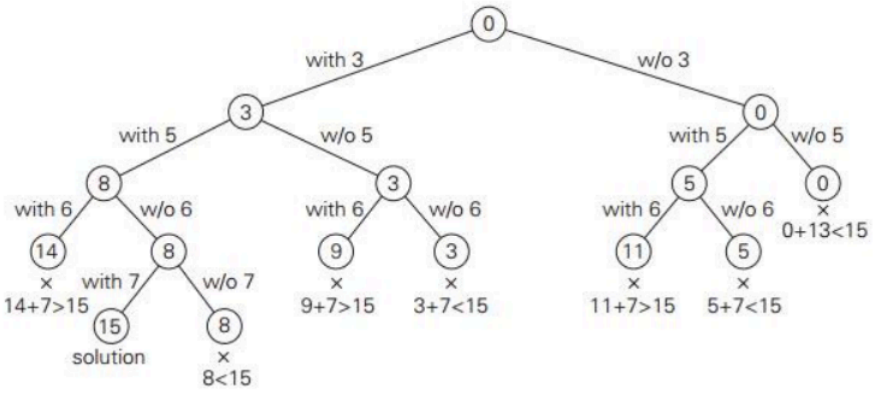
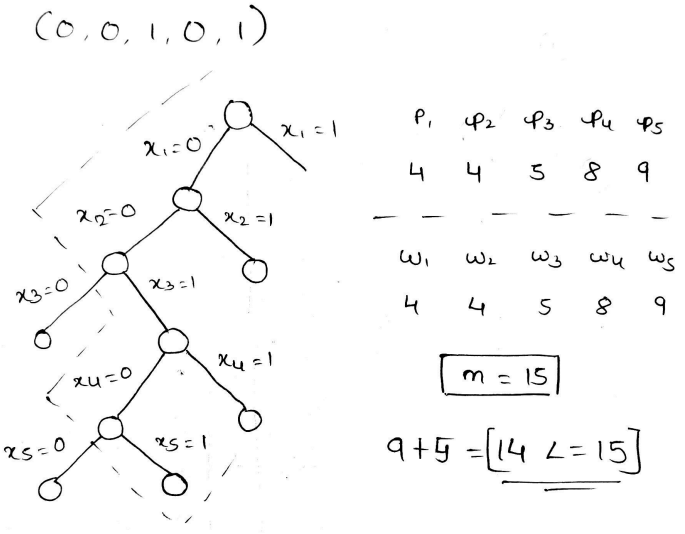


FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

Subsets do not add up to 11. Question seems incorrect. The above solution doesn't resonate with the above problem.

13) Construct the portion of the state space tree generated by LCBB for the knapsack instance: n=5, (p1, p2, p3, p4, p5) = (w1 w2, w3, w4, w5) = (4, 4, 5, 8, 9) and m=15.



14) Explain an algorithm for 4-queens problem using backtracking

Refer Part B 11th q

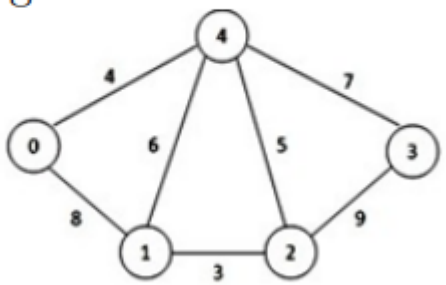
15) Apply Backtracking technique to solve the following instance for the subset problem s=(6,5,3,7) and d=15.

Same as 12th question

16) Build the portion of state space tree generated by FIFOBB for the job sequencing with deadlines instance n=5 , (p1,p2,..,p5) =(6,3,4,8,5), (t1,t2,..t5) = (2,1,2,1,1) and (d1,d2,..,d5)=(3,1,4,2,4). What is the penalty corresponding to an optimal solution?

17) Solve the solution for 0/1 knapsack problem using dynamic programming N=3 , m=6 profits (p1,p2,p3) = (1,2,5) weights (w1,w2,w3) = (2,3,4)

Refer this link **:** [0/1 Knapsack Problem | Dynamic Programming | Example | Gate Vidyalay]

18	Choose shortest distances using all pairs shortest path algorithm 

19) Solve knapsack problem by Dynamic Programming method n=6, (p1, p2,...p6)= (w1,w2,...w6)= (100,50,20,10,7,3) and m=165.

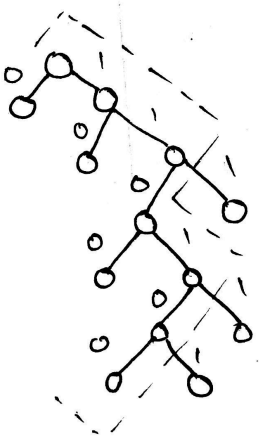
profits = weights

100. 50 20 10 7 3

m = 165 ; n = 6

100 + 50 + 10 = 160

1 1 0 1 0 0



20) Define greedy methods.

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Advantages of Greedy Approach

- The algorithm is easier to describe.
- This algorithm can perform better than other algorithms (but, not in all cases).

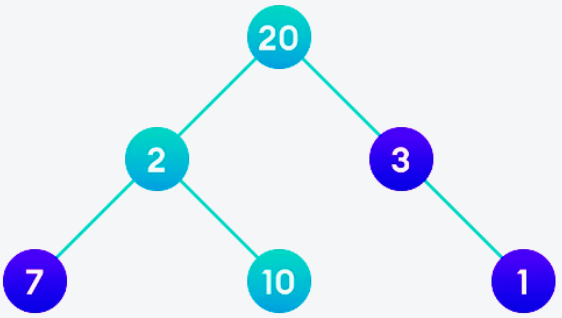
Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.

2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.

3. Finally the weight of an only child of **3** is **1**. This gives us our final result `20 + 3 + 1 = 24`.

However, it is not the optimal solution. There is another path that carries more weight (`20 + 2 + 10 = 32`) as shown in the image below.



Longest path

Few Pending

DAA MODULE 5

Click the link below to watch overview on Module 5

P, NP, NP Hard and NP Complete Problem | Reduction | NP Hard and NP C...

PART - A

1. Show that satisfiability is at most three literals reduced to chromatic number.

Refer - 3-coloring is NP Complete - GeeksforGeeks

4. Explain two problems that have polynomial time algorithms. Justify your answer.

A polynomial-time algorithm is an algorithm whose execution time is either given by a polynomial on the size of the input, or can be bounded by such a polynomial. Problems that can be solved by a polynomial-time algorithm are called tractable problems.

For example, most algorithms on arrays can use the array size, n, as the input size. To find the largest element in an array requires a single pass through the array, so the algorithm for doing this is O(n), or linear time.

Polynomial time	Exponential Time
n - Linear Search logn - Binary Search n*n - Insertion Sort n*logn - Merge Sort n*n*n - Matrix Multiplication	2^n - 0/1 knapsack 2^n - Travelling SP 2^n - Sum of Subsets 2^n - Graph Coloring 2^n - Hamilton Cycle

write any polynomial time problem

5. Explain 3CNF satisfiability problems.

Concept: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants

Such as (X+Y+Z) (X+Y+Z) (X+Y+Z)

You can define as (XvYvZ) ^ (XvYvZ) ^ (XvYvZ)

V=OR operator

^ =AND operator

These all the following points need to be considered in 3CNF SAT.

To prove: -

1. Concept of 3CNF SAT
2. SAT≤ρ 3CNF SAT
3. 3CNF≤ρ SAT
4. 3CNF ϵ NPC

1. CONCEPT: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
2. SAT ≤ρ 3CNF SAT:- In which firstly you need to convert a Boolean function created in SAT into 3CNF either in POS or SOP form within the polynomial time

```
F=X+YZ
    = (X+Y) (X+Z)
    = (X+Y+ZZ') (X+YY'+Z)
    = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
    = (X+Y+Z) (X+Y+Z') (X+Y'+Z)
```

3. 3CNF ≤p SAT: - From the Boolean Function having three literals we can reduce the whole function into a shorter one.

```
F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)
    = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
    = (X+Y+ZZ') (X+YY'+Z)
    = (X+Y) (X+Z)
    = X+YZ
```

4. 3CNF ϵ NPC: - As you know very well, you can get the 3CNF through SAT and SAT through CIRCUIT SAT that comes from NP.

   Proof of NPC:-

1. It shows us that we can easily convert a Boolean function of SAT into 3CNF SAT and satisfy the concept of 3CNF SAT also within polynomial time through Reduction concept.
2. If we want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

As we have achieved these two points that means 3CNF SAT also in NPC

6. Explain P type problems with examples.

Say that an algorithm A runs in polynomial time if there is a positive integer k so that,

A runs in time $O(n^k)$, where n is the length of x.

For example, an algorithm that takes no more than n^3 steps runs in polynomial time, or is a polynomial time algorithm.

If we accept, for the moment, that a polynomial time algorithm is "efficient", then it makes sense to give a name to decision problems that have efficient algorithms.

➔ P is the class of all decision problems (languages) that can be solved by polynomial time algorithms.

Notice that the definition has nothing to do with whether you or I are clever enough to find a polynomial time algorithm for the problem. A decision problem is in P if there exists a polynomial time algorithm for it.

Examples:

1. Recognizing palindromes:

The problem of recognizing palindromes is solvable in linear time, which is certainly polynomial time. A palindrome is a string that is equal to its own reversal. For example, aabcbaa is a palindrome. Let's define

```
PALINDROME =   {x | x ∈ {a, b, c}* and x is a palindrome}
```

It is easy to see that PALINDROME is in P. To decide if x is a palindrome, just reverse x and check whether the reversal of x is equal to x.

2. String matching

The string matching problem is as follows.

Input. Two strings, p and t.

Question. Does p occur as a contiguous substring of t?

For example if p = rocket and t = The rocketry expert is here., then the answer is yes.

One algorithm just tries each position in t to see if p occurs at that position. It keeps going until it finds a match or exhausts the positions that it can try.

The rocketry expert is here.

rocket

rocket

rocket

rocket

rocket

Suppose that p has length u and t has length v. There are v – u + 1 positions to try. It takes no more than u steps to check each position. So the total time is proportional to u(v – u + 1).

Since n ≈ u + v, the cost is highest when v is about 2u, which means u ≈ n/3. Then the time is about (n/3)(n/3), which is $O(n^2)$.

7. Explain in detail about approximation algorithms for NP hard problems.

Firstly let's discuss different approaches to handle difficult problems of combinatorial optimization, such as the travelling salesman problem and the knapsack problem.
The decision versions of these problems are NP-complete. Their optimization versions fall in the class of NP-hard problems. Theoretically there are no known polynomial-time algorithms for these problems.

There is a radically different way of dealing with difficult optimization problems: solve them approximately by a fast algorithm. This approach is particularly appealing for applications where a good but not necessarily optimal solution will suffice. Besides, in real-life applications, we often have to operate with inaccurate data to begin with. Under such circumstances, going for an approximate solution can be a particularly sensible choice.

Although approximation algorithms run a gamut in level of sophistication, most of them are based on some problem-specific heuristic. A heuristic is a common-sense rule drawn from experience rather than from a mathematically proved assertion.

We use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution s_a to a problem of minimising some function f by the size of the relative error of this approximation,

However, we cannot compute the accuracy ratio, because we typically do not know f (s∗), the true optimal value of the objective function. Therefore, our hope should lie in obtaining a good upper bound on the values of $r(s_a)$.
A polynomial-time approximation algorithm is said to be a c-approximation algorithm, where c ≥ 1, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question:

$$r(s_a) \leq c. \qquad \textbf{(12.3)}$$

The best (i.e., the smallest) value of c for which inequality (12.3) holds for all instances of the problem is called the performance ratio of the algorithm and denoted RA.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with RA as close to 1 as possible.

Note: Unfortunately, some approximation algorithms have infinitely large performance ratios (RA = ∞). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

8. Show that satisfiability of Boolean formula in 3 conjunctive normal form is NP - complete.

Concept: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants

Such as (X+Y+Z) (X+Y+Z) (X+Y+Z)

You can define as (XvYvZ) ^ (XvYvZ) ^ (XvYvZ)

V=OR operator

^ =AND operator

These all the following points need to be considered in 3CNF SAT.

To prove: -

5. Concept of 3CNF SAT
6. SAT≤ρ 3CNF SAT
7. 3CNF≤ρ SAT
8. 3CNF ϵ NPC
9. CONCEPT: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
10. SAT ≤ρ 3CNF SAT:- In which firstly you need to convert a Boolean function created in SAT into 3CNF either in POS or SOP form within the polynomial time

    F=X+YZ

    = (X+Y) (X+Z)

    = (X+Y+ZZ') (X+YY'+Z)

    = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)

    = (X+Y+Z) (X+Y+Z') (X+Y'+Z)
11. 3CNF ≤p SAT: - From the Boolean Function having three literals we can reduce the whole function into a shorter one.

```
F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)
    = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
    = (X+Y+ZZ') (X+YY'+Z)
    = (X+Y) (X+Z)
    = X+YZ
```

12. 3CNF ϵ NPC: - As you know very well, you can get the 3CNF through SAT and SAT through CIRCUIT SAT that comes from NP.

    Proof of NPC:-

3. It shows us that we can easily convert a Boolean function of SAT into 3CNF SAT and satisfy the concept of 3CNF SAT also within polynomial time through Reduction concept.
4. If we want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

As we have achieved these two points that means 3CNF SAT also in NPC

9. Show that the Clique Decision problem is NP - Complete.

To Prove: - Clique is an NPC or not?

For this we have to satisfy the following below-mentioned points: -

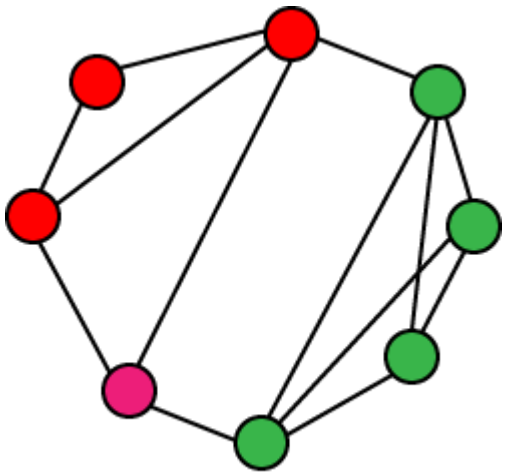
1. Clique
2. 3CNF ≤ρ Clique
3. Clique ≤ρ 3CNF≤SAT
4. Clique ϵ NP

1) Clique

Definition: - In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

CLIQUE COVER: - Given a graph G and an integer k, can we find k subsets of verticesV$_1$, V$_2$...V$_K$, such that UiVi = V, and that each Vi is a clique of G.

The following figure shows a graph that has a clique cover of size 3.



Clique cover (size=3)

2)3CNF ≤ρ Clique

Proof:-For the successful conversion from 3CNF to Clique, we have to follow the two steps:-

Draw the clause in the form of vertices, and each vertex represents the literals of the clauses.

1. They do not complement each other
2. They don't belong to the same clause
   In the conversion, the size of the Clique and size of 3CNF must be the same, and you successfully converted 3CNF into Clique within the polynomial time

3) Clique ≤ρ 3CNF≤SAT

Proof: - As we know that a function of K clause, there must exist a Clique of size k. It means that P variables which are from the different clauses can assign

the same value (say it is 1). By using these values of all the variables of the CLIQUES, you can make the value of each clause in the function is equal to 1

Example: - You have a Boolean function in 3CNF:-

(X+Y+Z) (X+Y+Z') (X+Y'+Z)

After Reduction/Conversion from 3CNF to CLIQUE, you will get P variables such as: - x +y=1, x +z=1 and x=1

Put the value of P variables in equation (i)

(1+1+0)(1+0+0)(1+0+1)

(1)(1)(1)=1 output verified

4) Clique ϵ NP:-

Proof: - As you know very well, you can get the Clique through 3CNF and to convert the decision-based NP problem into 3CNF you have to first convert into SAT and SAT comes from NP.

So, I concluded that CLIQUE belongs to NP.

Proof of NPC:-

1. Reduction achieved within the polynomial time from 3CNF to Clique
2. And verified the output after Reduction from Clique To 3CNF above
   So, concluded that, if both Reduction and verification can be done within the polynomial time that means Clique also in NPC.

10. Show that the Chromatic number Decision problem is NP - Complete.

PART - B

Bhavani, Rishi and Ujjwal

1. Explain and prove Cook's theorem.

In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

Stephen Arthur Cook and L.A. Levin in 1973 independently proved that the satisfiability problem(SAT) is NP-complete. He proved Circuit-SAT and 3CNF-SAT problems are as hard as SAT. Similarly, Leonid Levin independently worked on this problem in the then Soviet Union. The proof that SAT is NP-complete was obtained due to the efforts of these two scientists. Later, Karp reduced 21 optimization problems, such as Hamiltonian tour, vertex cover, and clique, to the SAT and proved that those problems are NP-complete.

Seems like some complex theorem. If interested, refer link below:
Design and Analysis Cook's Theorem

2. Compare deterministic and non deterministic algorithms.

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
1	Definition	The algorithms in which the result of every algorithm is uniquely defined are known as the	On other hand, the algorithms in which the result of every algorithm is not

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
		Deterministic Algorithm. In other words, we can say that the deterministic algorithm is the algorithm that performs a fixed number of steps and always gets finished with an accept or reject state with the same result.	uniquely defined and the result could be random are known as the Non-Deterministic Algorithm.
2	Execution	In Deterministic Algorithms execution, the target machine executes the same instruction and results the same outcome which is not dependent on the way or process in which the instruction gets executed.	On other hand in case of Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subject to a determination condition to be defined later.
3	Type	On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for	On other hand Non deterministic algorithms are classified as non-reliable

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
		a particular input instruction the machine will always give the same output.	algorithms for a particular input the machine will give different output on different executions.
4	Execution Time	As outcome is known and is consistent on different executions so the Deterministic algorithm takes polynomial time for their execution.	On other hand as outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.
5	Execution path	In deterministic algorithms the path of execution for an algorithm is the same in every execution.	On the other hand in case of a Non-Deterministic algorithm the path of execution is not the same for algorithm in every execution and could take any random path for its execution.

3. Explain non deterministic algorithms for sorting and searching.

In computer programming, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviours on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition.

The design of ND Algorithms is based on three major functions:

1. Select ()
2. Success ()
3. Failure ()

To declare the success or failure, a verification process should be designed.

```
Algorithm nd_search(a,n,x)
{
// "a" = array of size "n" and "x" is element to be searched
for i = 1 to n do
{
j = select(a,n) //select a location "j" from given array
if (a[j] = x)
success();
}
failure();
}
//Try above algorithm using repeat until and comment upon the time
complexity//
Algorithm nd_sort(a,b,n)
{
//"a" is array to be sorted and "b" is array for auxiliary storage
For i = 1 to n do
{
```

```
j = select(a,n)
b[i] = a[j] //create the array "b" by selecting "n" elements
}
for j = 1 to n do
{
if(b[i] > b[i+1])
failure();
}
success();
}
```

4. Explain non deterministic algorithm for sorting non deterministic knapsack algorithm.

```
S = empty ; total_value = 0 ; total_weight = 0 ; FOUND = false ;
Pick an order L over the objects ;
Loop
  Choose an object O in L ; Add O to S ;
  total_value = total_value + O.value ;
  total_weight = total_weight + O.weight ;
  If total_weight > CAPACITY Then  fail
  Else If total_value ≥ QUOTA
        FOUND = true ;
        succeed ;
  Endif Endif
  Delete all objects up to O from L ;
 Endloop
```

5. explain how P and NP problems are related

P-Class

- The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time O(n^k) in worst-case, where k is constant.
- These problems are called tractable, while others are called intractable or superpolynomial.
- Formally, an algorithm is polynomial time algorithm, if there exists a polynomial p(n) such that the algorithm can solve any instance of size n in a time O(p(n)).
- Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n. Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k.
- The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial slow-d.

NP-Class

- The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information.
- Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.
- Every problem in this class can be solved in exponential time using exhaustive search.

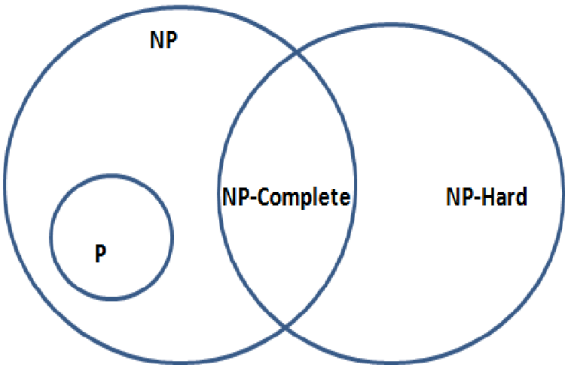
P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in NP - P are intractable.

It is not known whether P = NP. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If P ≠ NP, there are problems in NP that are neither in P nor in NP-Complete.
The problem belongs to class P if it's easy to find a solution for the problem. The problem belongs to NP, if it's easy to check a solution that may have been very tedious to find.



6. compare NP -hard and NP -complete problems.

NP Problem:

The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.

NP-Hard Problem:

A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

NP-Complete Problem:

A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

NP-hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, it does not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Do not have to be a Decision problem.	It is exclusively a Decision problem.
Example: Halting problem, Vertex cover problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, Circuit-satisfiability problem, etc.

7. Explain clique decision problems with an example.

In the field of computer science, the clique decision problem is a kind of computation problem for finding the cliques or the subsets of the vertices which when all of them are adjacent to each other are also called complete subgraphs.

A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other, that is the subgraph is a complete graph. The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices. This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph

or not. The clique decision problem has many formulations based on which the cliques and about the cliques the information should be found. There are some common formulations based on which the cliques are based such as finding the maximum clique, finding the maximum weight of the clique in a weighted graph, then listing all the maximum or maximal cliques, and finally solving the problem based on the decision of testing whether the graph has the larger cliques than that of the given size.

Maximum clique: a particular clique that has the largest possible number of vertices.

Maximal cliques: the cliques which further cannot be enlarged.

```
When S = NULL.
for i = 1 till k then start do-while loop.
t : = ch(1 to n)
if t belongs to S then
return fail.
When S:= S union t
Then for all pairs of I and j such that when i belongs to S and j belongs to S  and if i not equal to j then start do-while loop. And check if I and j is not an edge of that given graph then
Return fail.
Else return a true value.
```

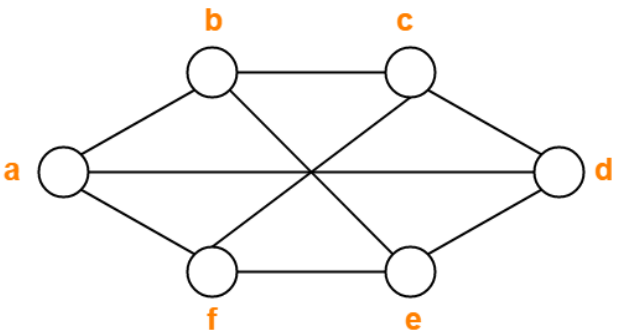
Conclusion :

Thus, a clique is actually a complete total subgraph of a given particular graph and the max clique association problem is basically a computational problem for finding the maximum clique of the graph.

8. Explain chromatic number decision problem and clique decision problem.

Chromatic Number is the minimum number of colors required to properly color any graph.

Consider the below example.



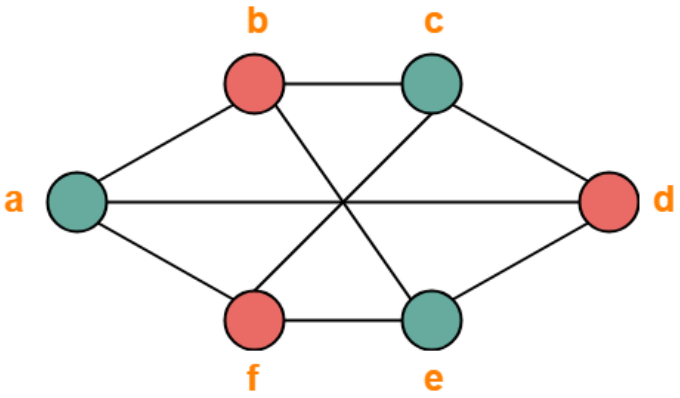
Applying Greedy Algorithm, we have-

Vertex	a	b	c	d	e	f
Color	C1	C2	C1	C2	C1	C2

From here,

- Minimum number of colors used to color the given graph are 2.
- Therefore, Chromatic Number of the given graph = 2.

The given graph may be properly colored using 2 colors as shown below-



Refer 7th Question

9. Explain the strategy to prove that a problem is NP - hard.

Reductions and SAT

To prove that any problem other than circuit satisfiability is NP-hard, we use a reduction argument. Reducing problem A to another problem B means

describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You've already been doing reduction for years, even before starting this book, only you probably called them something else, like subroutines or utility functions or modular programming or using a calculator. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.
To prove that your problem is hard, you need to describe an efficient algorithm to solve a different problem, which you already know is hard, using an hypothetical efficient algorithm for your problem as a black-box subroutine. The essential logic is a proof by contradiction. The reduction implies that if your problem were easy, then the other problem would be easy, which it ain't. Equivalently, since you know the other problem is hard, the reduction implies that your problem must also be hard;

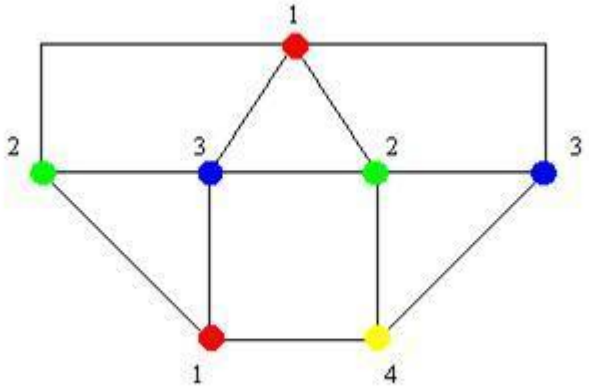
10. Explain intractable problems with examples.

- From a computational complexity stance, intractable problems are problems for which there exist no efficient algorithms to solve them.
- Most intractable problems have an algorithm – the same algorithm – that provides a solution, and that algorithm is the brute-force search.
- This algorithm, however, does not provide an efficient solution and is, therefore, not feasible for computation with anything more than the smallest input.
- The reason there are no efficient algorithms for these problems is that these problems are all in a category which I like to refer to as "slightly less than random." They are so close to random, in fact, that they do not as yet allow for any meaningful algorithm other than that of brute-force.
- If any problem were truly random, there would not be even the possibility of any such algorithm.

As the size of the integers (i.e. the size of n) grows linearly, the size of the computations required to check all subsets and their respective sums grows

exponentially. This is because, once again, we are forced to use the brute-force method to test the subsets of each division and their sums.

EXAMPLE : Graph colouring: How many colors do you need to color a graph such that no two adjacent vertices are of the same color?



Given any graph with a large number of vertices, we see that we are again faced with resorting to a systematic tracing of all paths, comparison of neighbouring colors, backtracking, etc., resulting in exponential time complexity once again.

11 - 16 Questions are as it is repeated from 1 - 6

17. What is the efficiency of Warshall's algorithm?

Warshall's Algorithm

Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix. For this, it generates a sequence of n matrices. Where, n is used to describe the number of vertices.

$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}$

A sequence of vertices is used to define a path in a simple graph. In the k^{th} matrix ($R^{(k)}$), ($r_{ij}^{(k)}$), the element's definition at the i^{th} row and j^{th} column will be

one if it contains a path from v_i to v_j. For all intermediate vertices, w_q is among the first k vertices that mean $1 \leq q \leq k$.

```
Warshall(A[1...n, 1...n])
// A is the adjacency matrix
R(0) ← A
for k ← 1 to n do
for i ← 1 to n do
for j ← 1 to n do
R(k)[i, j] ← R(k-1)[i, j] or (R(k-1)[i, k] and R(k-1)[k, j])
return R(n)
```

- Time efficiency of this algorithm is (n^3)
- In the Space efficiency of this algorithm, the matrices can be written over their predecessors.
- $\Theta(n^3)$ is the worst-case cost. We should know that the brute force algorithm is better than Warshall's algorithm. In fact, the brute force algorithm is also faster for a space graph.

18. Define the principle of backtracking.

Backtracking is an algorithmic technique whose goal is to use brute force to find all solutions to a problem. It entails gradually compiling a set of all possible solutions. Because a problem will have constraints, solutions that do not meet them will be removed.

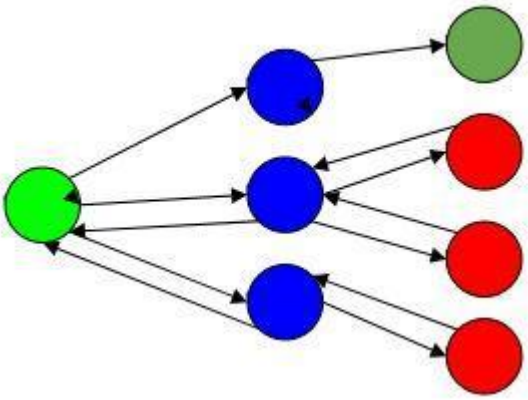
Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that don't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.

- Enumeration problem used to find the set of all feasible solutions of the problem.

In a backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.



Here,

Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is the end solution.

Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find the next point to find solution.

18. Define control abstraction for backtracking.

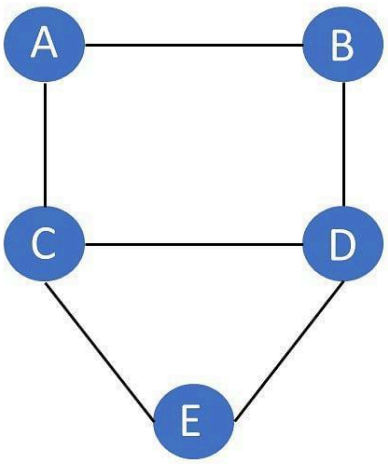
Control abstraction for backtracking defines the flow of how it solves the given problem in an abstract way.

In backtracking, solution is defined as n-tuple $X = (x_1, x_2, ..., x_n)$, where $x_i = 0$ or 1. x_i is chosen from set of finite components S_i. If component x_i is selected then set $x_i = 1$ else set it to 0. Backtracking approach tries to find the vector X such that it maximizes or minimizes certain criterion function $P(x_1, x_2, ..., x_n)$.

19. List the applications of backtracking.

1. To Find All Hamiltonian Paths Present in a Graph.

A Hamiltonian path, also known as a Hamilton path, is a graph path connecting two graph vertices that visit each vertex exactly once. If a Hamiltonian way exists with adjacent endpoints, the resulting graph cycle is a Hamiltonian or Hamiltonian cycle.



Hamilton Path: ABCDE

2. To Solve the N Queen Problem.

- The problem of placing n queens on the nxn chessboard so that no two queens attack each other is known as the n-queens puzzle.
- Return all distinct solutions to the n-queens puzzle given an integer n. You are free to return the answer in any order.
- Each solution has a unique board configuration for the placement of the n-queens, where 'Q' and. '' represent a queen and a space, respectively.

3. Maze Solving Problems

There are numerous maze-solving algorithms, which are automated methods for solving mazes. The random mouse, wall follower, Pledge, and Trémaux's algorithms are used within the maze by a traveller who has no prior knowledge of the maze. In contrast, a person or computer programmer uses the dead-end filling and shortest path algorithms to see the entire maze at once.

4. The Knight's Tour Problem

The Knight's tour problem is the mathematical problem of determining a knight's tour. A common problem assigned to computer science students is to

write a program to find a knight's tour. Variations of the Knight's tour problem involve chess boards of different sizes than the usual n x n irregular (non-rectangular) boards.