Name : Mangesh Khode
Class : B.Tech
Subject : DAA

Q1 Write short notes on the following :-

1) What is big O notation ?

Ans: The function $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$.

$f(n) = O(g(n))$

2) Differentiate linear search and binary search ?

Ans :

Linear search	Binary Search
* In linear search input data need not to be in sorted.	In Binary search input data need to be in sorted order.
* It is also called sequential search	It is also called half-interval search.
* The time complexity of linear search $O(n)$	Time complexity - $O(\log n)$
* It is very slow process	It is very fast process.
* Multidimensional array can be used.	only single dimensional array is used.

3) Why does the complexity of an algorithm need to be analyzed?

Ans: Analyzing the complexity of an algorithm is important for several reasons:

(1) Efficiency: The complexity analysis of an algo. helps to determine its efficiency, which is critical in optimizing its performance. By analyzing the complexity, we can determine how long it will take to execute the algo., how much memory it will require and how many resources it will consume.

(2) Comparision: Complexity analysis allows us to compare different algo. and determine which one is more efficient for a particular task. This helps in selecting the most appropriate algo. for solving a specific problem.

(3) Scalability: The complexity analysis of an algo. also helps to determine how it will perform as the size of the input grows. It enables us to predict how the algo. will scale, whether it will be able to handle larger inputs or not, and how it will perform as the problem size grows.

(4) Optimization: Complexity analysis is also crucial in optimizing algo. It helps to identify the parts of the algo. that are consuming the most resources and provides guidance on how to optimize them.

(5) Predictability: Complexity analysis allows us to predict the worst-case scenario for an algo. performance. This is important because it helps to ensure that the algo. will perform acceptably under any input size.

What do you mean by space complexity and time complexity?

: Space Complexity :- The space complexity of an algorithm is the amount of memory it needs to run to completion.

Space complexity is defined as the process of determining a formula for prediction of how much memory space will be required for the successful execution of the algo.

Time Complexity :- The time complexity of an algo is the amount of computer time it needs to run to completion.

The time complexity is the sum of the compile time and the run time.

4 Discuss the working of Merge-Sort technique with an example. Also explain its complexity.

Merge Sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray and then merging the sorted subarrays back together to form the final sorted array.

Merge Sort working Process :-

To known the functioning of merge sort lets Consider an array arr[] = { 38, 27, 43, 3, 9, 82, 10 }

* At first, check if the left index of array is less than the right index. if yes then calculate its mid point.

l = Left index r = Right index

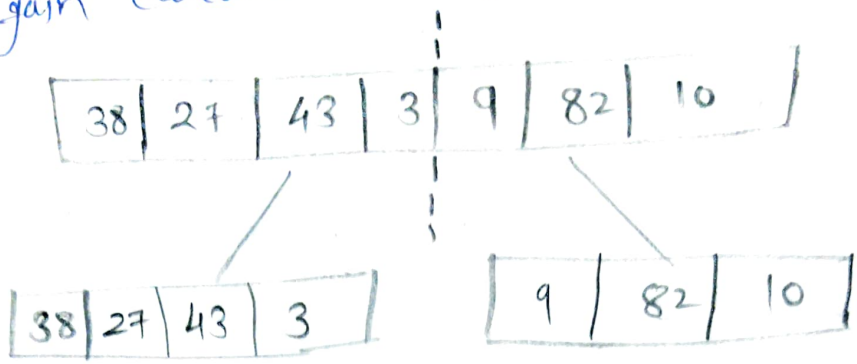
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

is l < r
Yes
m = l + (r-1)/2

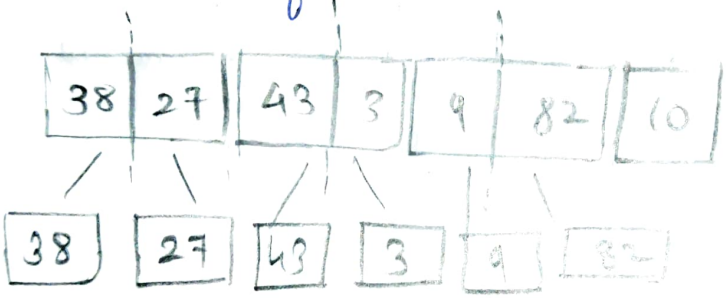
* Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic value are achieved.
* Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

| 38 | 27 | 43 | 3 | | 9 | 82 | 10 |

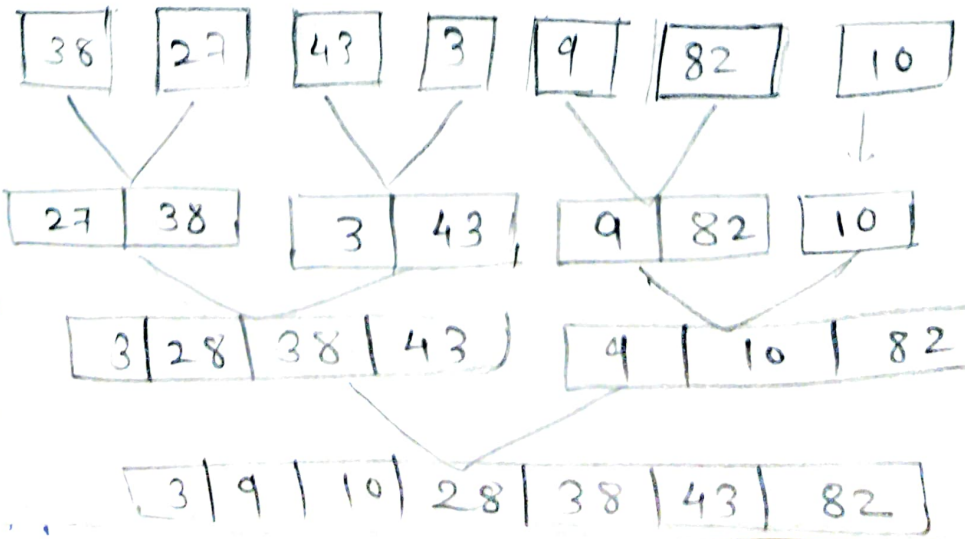
* Now, again find that is left index is less than right index for both arrays, If found yes, then again calculate mid points for both the array



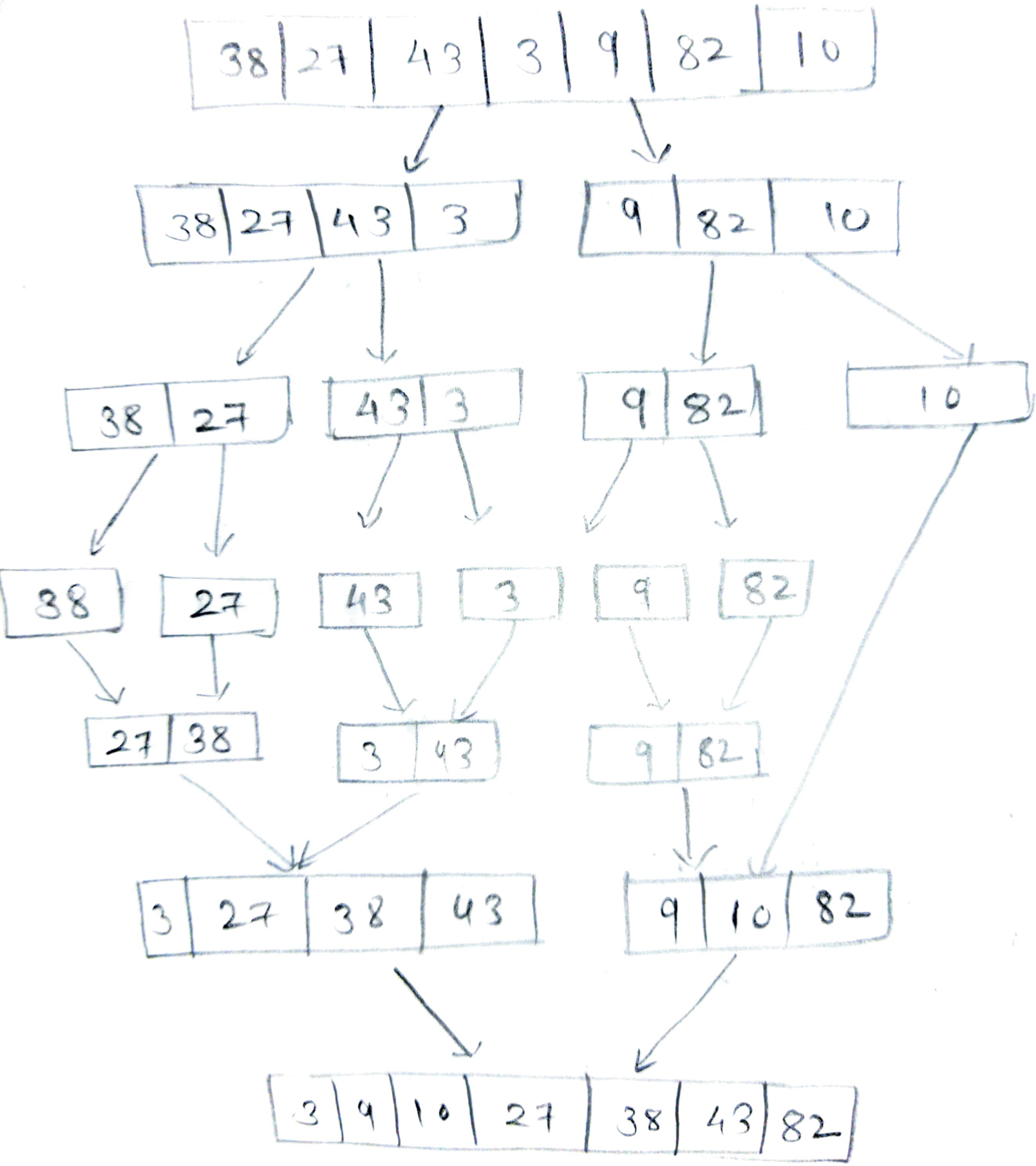
* Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



* After dividing the array into smallest units, start merging the elements again based on comparison of size of elements.
* Firstly, compare the elements for each list and then combine them into another list in a sorted manner.



* After the final merging, The list look like this →



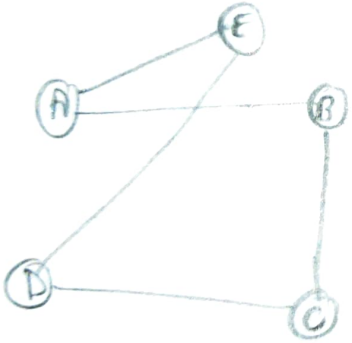
* Recursive steps of merge sort

What are the various methodes for Graph representation and its traversal? Explain with example.

Graph is a data structure to stores its data.
A Graph is a data structure (V, E) that consists of a collection of vertices V, and a collection of edges E. Graph Representation :-
A graph can be represented using 3 data structure adjacency matrix, adjacency list, and adjacency set.

① Adjacency matrix :- An adjacency matrix can be thought of as a table with rows and columns. The row labels and ~~with~~ ~~rows~~ ~~and~~ column labels represent the nodes of a graph.

eg.

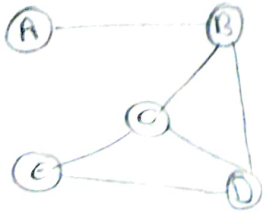


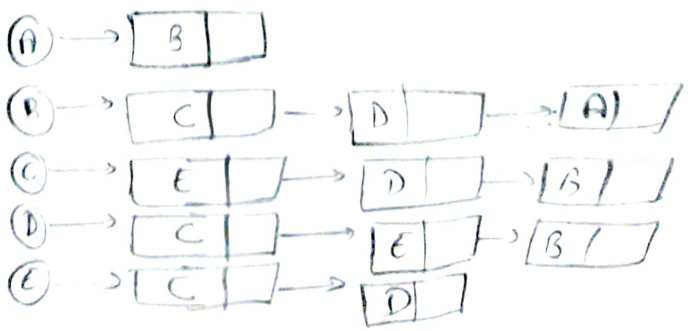
	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Adjacency matrix for an undirected graph

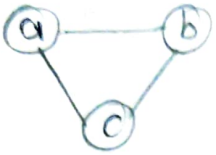
② Adjacency list :- It is a linked representation. In this representation, for each vertex in the graph, we maintain the list of its neighbors.

eg.





③ Adjacency list :- It is a collection of unorder lists used to represent a finite graph. Each unordered list within an adjacency list describes the set of neighbors of a particular vertex in the graph.



This undirected cyclic graph can described by the three unordered lists {b, c}, {a, c}, {a, b}.

Graph Traversal :- The process of visiting every node in the graph is called graph Traversal. There are 2 standard methods of graph traversal Breadth-First Search and Depth First Search.

BFS :- The breadth-first-search algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches / visits all nodes at the current depth level before moving on to the nodes at the next depth level.

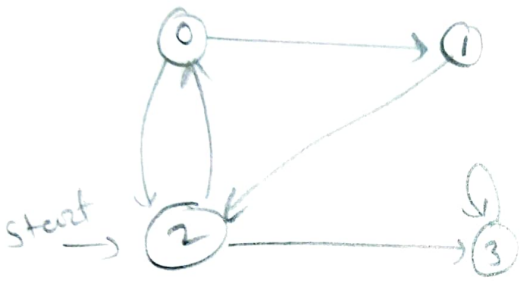
Algorithm :-

1) Consider the graph you want to navigate.
2) Select any vertex in your graph. Say v1, from which you want to traverse the graph.
3) Examine any two data structure for traversing the graph.
   Visited array (size of the graph)
   Queue data structure.
4) Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

5) Now, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.

6) Repeat step 5 until the queue is not empty and no vertex is left to be visited.

Example,

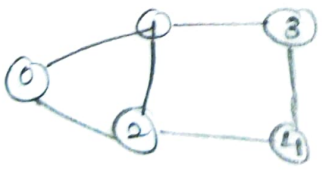


① we start traversal from vertex 2.

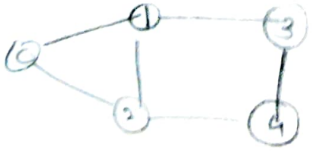
② when we come to vertex 0, we look for all adjacent vertices of it
* 2 is also an adjacent vertex of 0
* If we don't mark visited, then 2 will be processed again and it will become a non-terminating process.

③ There can be multiple BFS traversals for a graph
Different BFS traversals for the above graph:
2, 3, 0, 1 and 2, 0, 3, 1

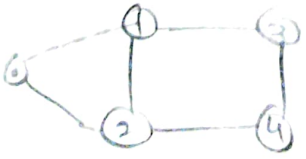
eg. :-



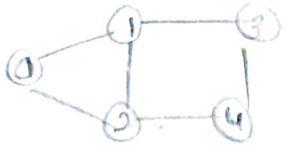
visited
queue
front



visited 0
queue 0
front



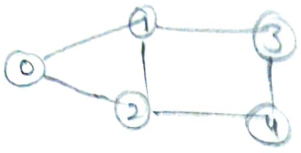
visited 0 1 2
queue 1 2



visited | 0 | 1 | 2 | 3 |

Queue | 2 | 3 |

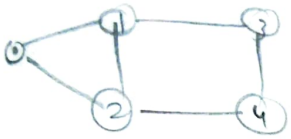
front



visited | 0 | 1 | 2 | 3 | 4 |

Queue | 3 | 4 |

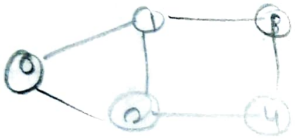
front



visited | 0 | 1 | 2 | 3 | 4 |

Queue | 4 |

front



visited [0 | 1 | 2 | 3 | 4]

Queue

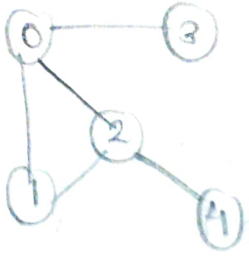
front

(2) Depth first search :

Algorithm :

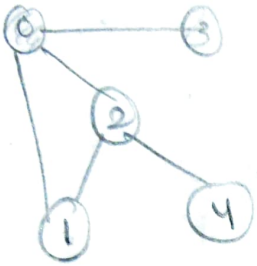
1) Start by putting any one of the graphs vertices on top of a stack.
2) Take the top item of the stack and add it to the visited list.
3) Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the sack.
4) Keep repeating steps 2 and 3 until the stack is empty.

eg.

① 

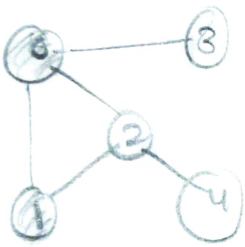
visited

stack

② 

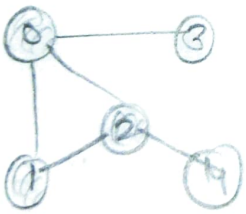
visited: 0

stack: 1 2 3

③ 

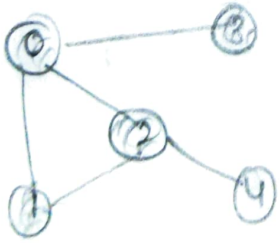
visited: 0 1

stack: 2 3

④ 

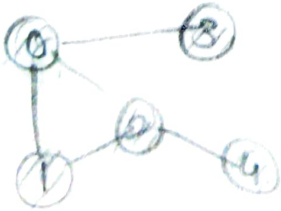
visited: 0 1 2

stack: 4 3

⑤ 

visited: 0 1 2 4

stack: 3

⑥ 

visited: 0 1 2 4 3

stack