**0:37**

**1. Introduction**

2. What is a data structure?

- data structures are different ways to organise data on your computer.

- before processing data, we must organise it to make it efficient. the efficiency of the program depends on how well the data is organised.

**0:09**

**1. Introduction**

3. What is an algorithm?

- an algorithm is a set of steps to perform a task.

- correctness and efficiency are the 2 main factors that make a good algorithm.

**0:26**

**1. Introduction**

5. Types of Data Structures

- 2 types of data structures, primitive and non-primitive data structures.

- primitive data structures are built into the program, such as integers, character, Boolean, float.

- on primitive data structures can be divided into linear and nonlinear data structures.

- linear data structures can be static or dynamic.

- examples of static data structures are array, examples of dynamic data structures are linked lists, stacks, queues.

- non-linear data structures include trees and graphs.

- each of these data structures have unique properties that are efficient in different circumstances.

**1:04**

**1. Introduction**

6. Types of Algorithms

- types of algorithms:

1. simple recursive algorithms

2. divide and conquer algorithms

3. dynamic programming algorithms

4. greedy algorithms

5. brute force algorithms

6. randomised algorithms

**0:11**

**2. Recursion**

10. What is Recursion?

- recursion is a way of solving a problem by making the function call itself.

- performing the same operation multiple times with different inputs.

- every time we make the problem smaller, it becomes easier to find the solution.

- the base condition is needed to fulfil the recursion of not it will become an infinite loop.

**1:05**

**2. Recursion**

11. Why do we need Recursion?

When to use recursion?

1. if you can divide the problem into smaller sub problems which are similar in nature

2. design an algorithm to compute nth

3. write code to list the n...

4. implement method to compute all

- recursion is used a lot in trees and graphs.

**0:18**

**2. Recursion**

12. The Logic Behind Recursion

- 2 conditions to recursion

1. a method calls itself.

2. exit from infinite loop.

- if the condition is met, exit from the loop, and return some value, if not method calls itself recursively.

- recursion is managed by stack memory, which works through last in first out (LIFO).

**0:22**

**2. Recursion**

13. Recursive vs Iterative Solution

- a conditional statement decides termination of recursion but in iteration a control value decides termination.

- recursion repeatedly invokes the mechanism consequently overhead of the method cost. this can be expensive in terms of processor time and memory space.

- if the problem can be divided into sub problems recursion is better.

- compared to iteration, recursion is not space efficient or time efficient but it’s easier to code.

**0:16**

**2. Recursion**

14. When to Use/Avoid Recursion?

- iteration beats recursion in terms of time and space complexity.

- recursion is better when you need a quick solution instead of an efficient one.

- recursion is better when we are traversing trees.

- recursion can be slow.

**1:00**

**5. Big O Notation**

26. What is Big O?

- Big O is the language and metric we use to describe the efficiency of algorithms.

- Time complexity is a way of showing how the runtime of a function increases as the size of input increases.

- Big O Notations:

1) Big O is a complexity that is going to be best or equal to the worst case. (Maximum time for algorithm execution.)

2) Big Omega is a complexity that is going to be at least more than the best case. (Minimum time for algorithm execution.)

3) Big Theta is a complexity that is going to be within the bounds of the best case and the worst case. (Average time for algorithm execution.)

**0:26**

**5. Big O Notation**

28. Most Common Time Complexities

- Complexities:

1) O (1)

- Name: Constant.

- Sample: Accessing a specific element in array.

2) O(N)

- Name: Linear.

- Sample: Loop through array of elements.

3) O(logN)

- Name: Logarithmic.

- Sample: Find element in a sorted array.

4) O(N^2)

- Name: Quadratic.

- Sample: Looking at every index in the array twice.

5) O(2^N)

- Name: Exponential.

- Sample: Double recursion in Fibonacci.

| Classes | n | *Complexity number of operations (10****)*** | Execution Time (1 instruction/μsec) |
| --- | --- | --- | --- |
| *constant* | *O(1)* | *1* | *1 μsec* |
| *logarithmic* | *O(logn)* | *3.32* | *3 μsec* |
| *linear* | *O(n)* | *10* | *10 μsec* |
| *O(nlogn)* | *O(nlogn)* | *33.2* | *33 μsec* |
| *quadratic* | *O(n2)* | *102* | *100 μsec* |
| *cubic* | *O(n3)* | *103* | *1msec* |
| *exponential* | *O(2n)* | *1024* | *10 msec* |
| *factorial* | *O(n!)* | *10!* | *3.6288 sec* |

****

**2:39**

**5. Big O Notation**

30. Drop Constants and Non-Dominant Terms

- drop constants and non-dominant terms.

- it is very possible that O(N) code is faster than O (1) code for specific inputs.

- different computers with different architectures have different constant factors.

- different algorithms with the same basic ideas and computational complexity might have slightly different constants.

Example: a\*(b-c) vs a\*b - a\*c.

**0:20**

**5. Big O Notation**

32. How to Measure the Code using Big O?

- How to measure codes using Big O.

Rule 1) Any assignment statements and if statements that are executed once regardless of the size of the problem. O (1)

Rule 2) A simple 'for' loop from 0 to n (with no internal loops) O(n)

Rule 3) A nested loop if the same type takes quadratic time complexity(n^2)

Rule 4) A loop in which the controlling parameter is divided by two at each step. O (log n)

Rule 5) When dealing with multiple statements just add them up.

**3:23**

**5. Big O Notation**

34. How to Measure Recursive Algorithm with Multiple Calls?

- for recursive function with multiple calls the runtime can be expressed as O (branches^ depth)

- for finding time complexity of recursive function initially, assume that the time complexity is M(N).

**2:24**

**6. Cracking Big O Interview Questions**

44. Question 10 - Time Complexity Powers Of 2

- the number of times we can half n before reaching 1 is O(log n).

**0:46**

**7. Arrays**

45. What is an Array?

- arrays can store data of specific types. an integer array cannot hold strings. different types of data cannot be stored in the same array.

- elements of the array are located next to each other. there is no gap between then in the memory.

- each array has a unique index. they can be identified or accessed based on the index.

- the size of array is pre- defined and cannot be changed.

- Definition: An array is a data structure consisting of a collection of elements, each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index using a mathematical formula.

**0:05**

**7. Arrays**

46. Types of Arrays

- Arrays can be divided into one dimensional arrays and multi-dimensional arrays.

- 1 D arrays have only a single row. A[i] will be between 0 and n where n is length of array.

- 2 D array is an array with a bunch of values having been declared with double index. A[i][j] means i and j between 0 and n. i is index of row and j s index of column.

- 3 D array is like 2 D array but instead of 2 parameters there are 3 to describe the position of the element in the array. a[i][j][k]. i, j and k are depth, row, column respectively.

- regardless of the number of dimensions the elements of the array are located continuously in the memory.

**0:21**

**7. Arrays**

48. Create an Array

- to create an array there are three steps:

Declare - creates a reference to array.

Instantiation of an array - creates an array.

Initialisation - assigns values to cells in array.

-to print out array into the console must import array library. import java.util.Arrays

**1:10**

**7. Arrays**

50. Accessing Elements in Array

-to access array element, <array name>[index]

- first element is at the 0 index.

- Array traversal: we are visiting all indexes in the array.

**0:11**

**7. Arrays**

55. Create Two-Dimensional Array

- for a 2D array in memory the array is reflected as a one-dimensional array, the first row is first, then second, etc.

- to print 2 D arrays use the deepToString function.

**1:18**

**7. Arrays**

58. Traverse Two Dimensional Array

- for 2D array traversal the logic is as such: first all elements of the first row are visited, then the second row, then the third row etc.

**0:01**

**7. Arrays**

62. When to Use/Avoid Arrays

-When to use arrays:

1. When there is a need to store multiple variables of the same data type.

2. Random access.

- When not to use arrays:

1. When u need to store data of different types.

2. Array useds specific amount of memory to hold its elements. so u cannot exceed the size of the array as it is fixed.

**0:02**

**11. Linked List**

83. What is a Linked List?

- Linked list is a form of a sequential collection and it does not have to be in order. A linked list is made up of independent nodes that may contain any type of data and each node has a reference to the next node in the link.

- Each node is independent.

- train analogy, Cars: passengers and links (nodes: data and references)

**0:08**

**11. Linked List**

84. Linked List vs Array

- Differences between linked lists and arrays:

1. Elements of linked lists are independent objects but in arrays, elements are not independent objects.

2. The size of linked lists are not predefined like arrays. so linked lists' size can be altered.

3. Random access - accessing an element in an array is very efficient, you just need to write the array name and index. But for linked lists you have to start from the head, then to the first node, then the second node etc, because the information of the location of the next node is stored in the current node. in linked lists.

**4:46**

**11. Linked List**

85. Types of Linked List

1. Singly Linked List

- a node does not have any reference to the previous node but has reference to the next node.

- in the first node the physical location of the second node is stored. so we only can go from second node to third node but cannot go from second node to the first node.

- the last node's reference is always null.

2. Circular Singly Linked List

- when we reach the last node of the linked list, we have the option to go back to the first node,  unlike in just a singly linked list. this is the only difference between the circular singly linked list and singly linked list.

3. Doubly Linked List

- unlike singly linked lists, we have references to the previous node and also to the next node.

4. Circular Doubly Linked List

- in the first node we have previous node reference and in the last node we have next node reference. this is the difference between circular doubly linked list and doubly linked list.

**2:18**

**11. Linked List**

86. Linked List in the Memory

- for linked list in memory, we cannot access any element directly, only from the previous node. the elements are randomly located in memory.

- the size of linked list can be altered after it is created.

**0:48**

**11. Linked List**

87. Creation of Singly Linked List

Creation of singly linked list:

1. First create head and tail, initialise with null.

2. Create a blank node and assign a value to it and reference to null.

3. Link head and tail with this node.

**2:19**

**13. Doubly Linked List**

120. Delete Entire Doubly Linked List

- for deletion of entire doubly linked list just traverse through the list and set all the prev references of all the nodes to null. then after that set the head reference and the tail reference to null. then all the nodes will become eligible for garbage collection.

**1:06**

**16. Stack**

144. What and Why of Stack?

- Stack is a data structure that stores items in last in first out manner.

**0:29**

**16. Stack**

145. Stack Operations

- to create a stack you just need to initialise an empty array or liquidly inside stack class without any element in it.

- push means insert, pop means remove.

-   peek method will return a value, starting from the top element in the stack. it does not remove it.

- isEmpty method returns true or false, to check of there are any elements in the stack or not.

- isFull method returns true or false, to check if the stack is full or not.

- delete method deletes all the elements in the stack.

**0:11**

**16. Stack**

146. Stack using Array vs Linked List

* Stacks using array

1. easy to implement
2. fixed size

* Stacks using linked list

1. variable size
2. implementation is not easy

**0:33**

**16. Stack**

153. When to Use/Avoid Stack

When to use or avoid stack

- use when there is LIFO functionality required.

- use when the chance of data corruption is minimum.

- avoid when random access is not possible.

**0:32**

**17. Queue**

155. What is a Queue?

- A queue is a data structure that stares items in a first in, first out manner.

**0:13**

**17. Queue**

156. Linear Queue Operations using Array

Queue operations

- create queue

- Enqueue

- Dequeue

- Peek

- isEmpty

- isFull

- deleteQueue

**0:18**

**17. Queue**

160. Why Circular Queue?

- we need circular queue because dequeue causes blank cells.

**0:10**

**19. Tree / Binary Tree**

181. What is a Tree?

- A tree is a non linear data structure with hierarchical relationships between its elements without having any cycle, it is basically reversed from a real life tree.

Properties:

- represent hierarchical data.

- each node has two components : data and a link to its sub category.

- base categories and sub categories as well.

**1:33**

**19. Tree / Binary Tree**

182. Why Tree?

Why use tree data structure?

- in order to perform any operation in linear data structures the time complexity increases with size.

-in non linear data structure like tree allows quicker and easier access to the data.

- store hierarchical data . like folder structure, organisation structure, XML/HTML data.

- there are many different types of data structures which performs better in various situations

- binary search tree, avl, red black tree, trie.

**0:33**

**19. Tree / Binary Tree**

183. Tree Terminology

Tree Terminology

- Root is the top node without parent.

- Edge is a link between parent and child.

- Leaf is a node which does not have children.

- Sibling is children of same parent.

- Ancestor is a parent, grandparent, great grandparent of a node.

- Depth of node is a length of the path from root to node.

- Height of node is a length of the path from the node to the deepest node.

- Depth of tree is the depth of root node.

- Height of tree is height of root node.

**0:24**

**19. Tree / Binary Tree**

185. What is A Binary Tree?

-Binary trees are the data structures in which each node has at most two children, often referred to as the left and right children.

- binary tree is a family of data structures. (BST, heap tree, AVL and Red Black trees.

- Huffman coding problem, heap priority problem, and expression parsing problems can be solved efficiently using binary trees.

**0:06**

**19. Tree / Binary Tree**

186. Types of Binary Tree

Types of binary trees

1. Full binary tree

- each node has 2 children or no children, no nodes have only 1 child. children to the left as possible.

2. Perfect binary tree

- all non leaf nodes that are located at the same depth have 2 children.

3. Complete binary tree

- all levels are completely filled except the last level, last level has nodes to the left of possible.

4. Balanced binary tree

- each leaf is not more than a certain distance from the root node than any other node.

**0:09**

**19. Tree / Binary Tree**

187. Binary Tree Representation

- A binary tree can be represented in 2 ways, via linked lists or arrays.

- for binary tree with array, left child = cell[2x], right child = cell[2x+1], x being the index of the parent.

**0:33**

**19. Tree / Binary Tree**

189. PreOrder Traversal in Binary Tree using Linked List

Binary tree traversal

1. depth first search

- preorder traversal

- inorder traversal

- post order traversal

2. breadth first search

- level order traversal

**1:29**

**19. Tree / Binary Tree**

190. InOrder Traversal in Binary Tree using Linked List

Orders of traversal for binary tree:

-preorder traversal: root node, left node, right node (keep left)

- inorder traversal: left subtree, root node, right subtree (keep left)

**1:40**

**19. Tree / Binary Tree**

191. PostOrder Traversal in Binary Tree using Linked List

Post Traversal:

- left subtree-> right subtree-> root node

**0:33**

**19. Tree / Binary Tree**

199. PreOrder Traversal Binary Tree (Array)

Binary tree array traversal:

Depth first search:

- preorder traversal -> visit root node then left subtree then right subtree.

- inorder traversal -> visit left subtree then root then right subtree.

- post order traversal -> visit left subtree then right subtree then root.

Breadth first search:

- level order traversal

**1:07**

**20. Binary Search Tree**

208. What is a Binary Search Tree? Why do we need it?

What is a binary search tree?

- in the left subtree, the value of a node is less than or equal to its parent node's value.

- in the right subtree the value of a node is greater than its parent node's value.

- it performs faster than binary tree when inserting and deleting nodes.

**1:08**

**21. AVL Tree**

220. What is an AVL Tree?

- an AVL tree is a self-balancing binary search tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

- to check of a tree is an avl one compare the heights of left and right subtrees for every node.

**0:09**

**21. AVL Tree**

222. Common Operations on AVL Tree

Common AVL Tree operations:

- creation of avl tree

- search for a node in avl trees

- traverse all nodes in avl trees

- insert a node in avl trees

-delete a node from avl trees

- delete the entire avl trees

**1:07**

**21. AVL Tree**

223. Insert a Node in AVL (Left Left Condition)

- 2 cases for insertion of node into AVL tree, rotation is required and rotation is not required.

- when rotation is required, figure out which condition it is. i.e. left left condition, left right condition, right right condition, right left condition. locate the grandchild of displaced node and figure out the condition through that.

- left left condition: do a right rotation.

- left right condition: do a left rotation first on left child on displaced node, then do a right rotation.

- right right condition: do a left rotation.

- right left condition: do a right rotation first on right child of displaced node, then do a left rotation.

**0:15**

**21. AVL Tree**

229. Delete a Node from AVL (LL, LR, RR, RR)

Deleting a node from avl tree:

- The node to be deleted is a leaf node.

- The node to be deleted has a child node. -> child node takes the place of deleted node.

- the node to be deleted has 2 child nodes -> the deleted node will be replaced by the successor node which is the minimum node in the right subtree of the deleted node.

First case: Rotation is not required

Second case: Rotation is required

- check if parent of deleted node is disbalanced or not -> if it is find the condition of the disbalanced node.

**0:27**

**22. Binary Heap**

236. What is Binary Heap? Why do we need Binary Heap?

A binary heap is a binary tree with the following properties.

- A binary heap is either min heap or max heap. the key at root must be minimum among all keys present in binary heap. The same property must be recursively true for all nodes in binary tree.

It's a complete tree (all levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of binary heap makes them suitable to be stored in an array.

-Common operations:

creation

peek(to find root of binary heap)

extract min

extract max

traversal

size

insert

delete

**0:32**

**23. Trie**

243. What is a Trie? Why we need Trie?

- A trie is a tree-based data structure that organises information in a hierarchy.

Properties:

- it is typically used to store or search strings in a space and time efficient way.

- Any node in the can store non repetitive multiple characters

- Every node stored link of the next character of the string

-Every node keeps track of "end of string"

Operations:

Creation

Insertion

Search for a string

Deletion

**11:34**

**23. Trie**

245. Insert a String in Trie

- if the character is present as a key in the current node it will map to the subsequent node. if it is not present, the value will return null.

- practical use of trie: spell checker, auto complete function in search engines.

**0:28**

**24. Hashing**

250. What is Hashing? Why we need it?

- Hashing is a method of sitting and indexing data. The idea behind hashing is to allow large amounts of data to be indexed using keys commonly created by formulas.

- A function converts strings to numbers using some formulas. By using these numbers, the strings will be inserted into the respective indexes equaling the numbers in an array.

- hashing is more efficient than other data structures for search operations.

- best case is O(1).

**0:18**

**24. Hashing**

251. Hashing Terminology

- Hash function is a function that can be used to map of arbitrary size to data of fixed size.

- Key is an input data given by a user.

- Hash value is a value returned by the hash function.

- Hash Table is a data structure which implements an associative array abstract data type, a structure that can map keys to values.

- Collision occurs when two different keys to a hash function produce the same output.

**6:29**

**24. Hashing**

252. Hash Functions

Characteristics of a good hash function:

- It distributes hash value uniformly across hash tables.

- It has to use all the input data.

**1:15**

**24. Hashing**

253. Types of Collision Resolution Techniques - Direct Chaining (Insert)

Resolution Techniques:

1. direct chaining

2. open addressing

- linear probing

- quadratic probing

- double hashing

Direct chaining: Implements the buckets as linked lists. Colliding elements are stored in this lists.

**3:27**

**24. Hashing**

255. Hash Table is Full

- Open addressing-> create 2X size of current hash table and recall hashing for current keys.

**0:31**

**24. Hashing**

256. Collision Resolution Technique - Linear Probing (Insert)

Open Addressing: Colliding elements are stored in other vacant beckets. During storage and lookup these are found through so called probing.

Linear Probing: It places new key into closest following empty cell.

- Decision to create new hash table is based on load factor, if it is greater than 0.75 a new hash table is created.

Quadratic probing: Adding arbitrary quadratic polynomial to the index until an empty cell is found.

Double Hashing: Interval between probes are computed by another hash function.

**1:26**

**24. Hashing**

260. Pros and Cons of Resolution Techniques

Direct chaining

- hash table never gets full

- huge linked list causes performance leaks (time complexity for search operation becomes O(N))

Open addressing

- easy implementation

- when hash table is full, creation of new hash table affects performance (time complexity for search operation becomes O(N).

- if the input size is known we always use open addressing. in this case the biggest problem of open addressing which is exhaustion of array cells will never happen.

- if we perform deletion operation frequently direct chaining is better.

**0:41**

**24. Hashing**

261. Practical Use of Hashing

Practical uses of hashing

- password verification (you can use key to get hash value but not hash value to get the key.

- file system -> file path is mapped to physical location on disk.

**0:39**

**25. Sorting Algorithms**

264. What is Sorting?

- By definition sorting refers to arranging data in a particular format : either ascending or descending.

Practical use of sorting:

- microsoft excel-> built in functionality to sort data

- online stores-> online shopping websites generally have option for sorting by price, review, ratings etc.

**0:47**

**25. Sorting Algorithms**

265. Types of Sorting

- Based on the spaced used algorithms can be in place or out place.

- In place sorting: sorting algorithms which does not require any extra space for sorting. example: bubble sort.

- out place sorting: sorting algorithms which requires an extra space for sorting. example: merge sort.

- Based on the stability algorithms can be stable or unstable.

- Stable sorting: if a sorting algorithm after sorting the contents does not change the sequence of similar content in which they appear, then this sorting is called stable sorting. example: insertion sort.

- Unstable sorting:; If a sorting algorithm after sorting the content changes the sequence of similar content in which they appear, then this sorting is called unstable sorting. example: quick sort.

**0:32**

**25. Sorting Algorithms**

266. Sorting Terminology

- increasing order means if a successive element is greater than the previous one.

- decreasing order means if a successive element is less than the previous one.

- non increasing order: if a successive element is less than or equal to its previous element in the sequence.

- non decreasing order: if a successive element is greater than or equal to its previous element in the sequence.

when choosing an algorithm must consider stability, time complexity and space complexity.

**0:41**

**25. Sorting Algorithms**

267. Bubble Sort

- bubble sort is also referred as sinking sort

- we repeatedly compare each pair of adjacent items and swap them if they are in the wrong order.

when to use bubble sort

- when the input is almost sorted

- space is a concern

- easy to implement

when to avoid bubble sort:

average time complexity is poor

**7:18**

**25. Sorting Algorithms**

269. Insertion Sort

When to use insertion sort

- when we have insufficient memory

- it is easy to implement

- when he have continuous inflow of numbers and we want to keep them sorted

when not to use insertion sort

- when time is a concern.

**1:19**

**25. Sorting Algorithms**

270. Bucket Sort

Bucket Sort

- create buckets and distribute elements of array into buckets

- sort buckets individually

- merge buckets after sorting

- number of buckets = round(sqrt(number of elements))

- appropriate bucket = celi(value\* number of buckets / maxValue)

when to use bucket sort

- when input is uniformly distributed over range.

when to avoid bucket sort

when space is a concern.

**0:11**

**25. Sorting Algorithms**

271. Merge Sort

- merge sort algorithm is a divide and conquer algorithm

- divide the input array into two halves and we keep halving recursively until they become too small that they cannot be broken further.

when to use merge sort

- when you need stable sort

- when average expected time is o(NlogN)

when to avoid merge sort

- when space is a concern.

**0:28**

**25. Sorting Algorithms**

273. Heap Sort

Heap sort

step 1: insert data to binary heap tree

step 2: extract data from binary heap

it is best suited with array, it does not work with linked list

**1:13**

**26. Searching Algorithms**

281. Time Complexity of Binary Search

- binary search is used for sorted arrays while linear can be used for unsorted arrays.

- binary search has average time complexity of O(logN) while time complexity of linear search is O(N).

**0:02**

**27. Graphs**

284. What is a Graph? Why do we need Graph

- a graph consists of a finite set of vertices (or nodes) and a set of edges which connect a pair of nodes.

**0:18**

**27. Graphs**

285. Graph Terminology

graph terminology

- vertices(vertex): vertices are the nodes of the graph

- edge: the edge is the line that connects pairs of vertices.

- unweighted graph: a graph which does not have a weight associated with any edge.

- weighted graph: a graph which has a weight associated with any edge

- undirected graph: in case the edges of the graph do not have a direction associated with them.

**0:57**

**27. Graphs**

286. Types of Graph

graph types

- unweighted - undirected.

- unweighted - directed.

- positive - weighted - undirected.

- positive - weighted - directed.

- negative - weighted - undirected. (a graph is negative if any weight in the graph is negative.

- negative - weighted - directed.

**0:27**

**27. Graphs**

287. Graph Representation

- adjacency matrix: an adjacency matrix is a square matrix or you can sat it is a 2D array. And the elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

- adjacency list: an adjacency list is a collection of unordered list used to represent a graph. each list describes a set of neighbours of a vertex in a graph.

- if the graph is complete or almost complete we should use an adjacency matrix. if the number of edges are few we should use an adjacency list.

**1:29**

**28. Graph Traversal - Breadth First Search and Depth First Search Algorithms**

291. Breadth First Search Algorithm (BFS)

- BFS is an algorithm for traversing graph data structure. It stares at some arbitrary node of a graph and explores the neighbour nodes(which are at current lever) first, before moving to the next level neighbours.

**0:22**

**28. Graph Traversal - Breadth First Search and Depth First Search Algorithms**

295. Depth First Search (DFS) Algorithm

DFS is an algorithm for traversing a graph data structure which starts selecting some arbitrary node and explores as far as possible along each edge before backtracking.

**2:10**

**28. Graph Traversal - Breadth First Search and Depth First Search Algorithms**

299. BFS Traversal vs DFS Traversal

- use depth first search if we know that the target vertex is already buried very deep.

- use breadth first search if we know that the target vertex is near to the starting point.

**1:04**

**29. Topological Sort**

301. What is Topological Sort?

- Topological sort sorts given actions in such a way that if there is a dependency of one action on another, then the dependent action always comes later than its parent action.

**0:33**

**30. Single Source Shortest Path Problem (SSSPP)**

307. What is Single Source Shortest Path Problem?

A single source problem is about finding a path between a given vertex (called source) to all other vertices in a graph such that the total distance between them (source and destination) is minimum.

**3:11**

**31. Dijkstra's Algorithm**

318. Dijkstra's Algorithm with Negative Cycle

-Dijkstra's algorithm does not work with negative cycles because the port times the cycle repeats the shorter the path.

**1:04**

**32. Bellman Ford Algorithm**

320. Bellman Ford Algorithm for SSSPP

- Bellman ford algorithm is used to find single source shortest path algorithm. If there is a negative cycle it catches and reports its existence.

**1:19**

**33. All Pairs Shortest Path Problem**

326. What is All Pairs Shortest Path Problem?

- All pair shortest path problem is about finding a path between every vertex to all other vertices in a graph such that the total distance between them (the source and the vertex) is at a minimum.

**1:53**

**34. Floyd Warshall Algorithm**

329. Why Floyd Warshall Algorithm?

Floyd Warshall Algorithm

- there are only three possible outcomes for the distance between vertices.

1. the vertex is not reachable.

2. the 2 vertices are directly connected.

3. the two vertices are connected via other vertex.

**0:33**

**35. Minimum Spanning Tree (Disjoint Set)**

335. What is Disjoint Set?

- A disjoint set is a data structure that keeps track of set of elements which are partitioned into a number of disjoint and non overlapping sets and each sets have representative which helps in identifying that sets.