# Data Structures and Algorithms:

**Data Structures + Algorithms = Programs**

* Algorithms: Set of instructions

**Logarithms : Flip of Exponents**

Logx y = How many x’s multiplied would give y

Log2 8 = 3. Three 2’s multiplied would give 8

Default base for any log is 2.

Data Structure – Way of structuring and organising data in a computer so that it can be used efficiently.  
Two types:  
Linear and Nonlinear.

It is wrto elements access, Elements can be accessed linearly or non-linearly. The Storage may or may not be linear.  
Data Structures + operations = Abstract data type.

Logarithmic complexity: Divide and Conquer. Eg- Binary Search.  
If input grows exponentially, i.e., 2^n. Time grows as n.

Search Algorithms:Binary Search:

Partition the data structure to be searched to two, and then guess the number. Take the midpoint as the base.

1. If the required number is greater than the midpoint, rearrange the search partition which starts from midpoint + 1 to the end.
2. If the required number is less than the midpoint, the search partition would be Beginning to midpoint-1
3. If the required number is equal to the midpoint, then you have the result, hence break.

* Binary search – Logarithmic time complexity because each time the partition is divided by 2. So in takes logn steps to divide the whole structure. Time complexity is the number of steps it takes to perform the operation for n items.
* Binary Search only works on sorted inputs.

# BigO Notation:

To quantify algorithm’s RATE of Performance.   
Not every algorithm grows at the same rate.

Rate = Runtime / Size of Input

Ignore the addition, subtraction, multiplication, division factors.

Highest BigO being n! for the Travelling Salesman problem.

Reason: For the salesman to travel to n cities, to know the shortest route he needs to know all the possible permutations of the routes possible, and then pick the shortest route of them.   
Permutations – nPr. Which n!/(n-r)!   
Because we ignore the division constants, the BigO is n!.

O(1) time complexity doesn’t mean it is instant, It means that irrespective of the size of the input, it takes constant time.

### Tip – Time Complexity:

If at any point of time, you are unable to find the time complexity, divide the problem into the simplest possible length, and find the steps it takes to solve the problem. Keep increasing the length by 1 and then find the steps again.

### Tip:

If it is given N <= 1 000 000, then the time complexity your solution should give is – O(n) or max O(logn)

If N < = 1 0000, then you can afford to go for O(n^2).

Computer Memory is like a chest of lockers wherein you request each locker depending on the size of items you want to store.

Arrays and Linked Lists trade-offs:

1. Arrays   
   pros – continuous memory – faster access of random elements – random access  
   Cons - Need a bunch at a time – if we pre-allocate the bunch – it is waste of memory.
2. Linked list   
   pros - no need of bunch – easy insertions and deletions  
   Cons - to read last item, parse through the whole list, sequential access

Insertions and deletions in LinkedList are O(1) time only if you can instantly access the element to be deleted. It’s a common practice to keep track of the first and last items in a linked list, so it would take only O(1) time to delete those.  
Using a hybrid datastructure – you can get best of both the worlds.

Arrays and Linked lists are building blocks of complex datastructures.

# Sorting Algorithms:

## Selection Sort:

Loop through the whole array and find the shortest. Select the shortest and then loop through the remaining to find the next shortest. Similarly n times. O(n) to find the shortest element and n times, so O(n\*n) – O(n\*2)

## Recursion:

All Divide and Conquer can be implemented using recursion, but not vice-versa.

Iterative algorithms need a stack data type to push and pop after each iteration, whereas recursion doesn’t need a stack as it uses the existing function call stack.

Recursion logic – every problem can be divided into sub problems, which is a problem in itself and the way to solve the subproblem is same as solving the main problem. Transform the problem into a self-equal representation.  
Any recursive approach can be transformed to an iterative approach using a loop and a stack.

Base case for an array is often and empty array or an array with one element.

Recursion Pros and Cons. Cons – It needs extra stack space. Infinite recursion – stack overflow.  
Pros – It’s easy to get into a recursive approach.

Recursion and Divide and Conquer work hand in hand. Apply recursion when you can divide a problem into similar subproblems.

Recursion is a Loop for which you need to know when to break the loop, when to return from the loop and how to continue the loop.

Break the loop - base condition

Continue the loop – recursion condition

/// Return from the loop – success condition --- Ideally base case is success condition

Tail Recursion – When recursion call is at the end, i.e., at the tail. It means that there are no more instructions after the recursion call.

**Euclidean Algorithm:**Greatest Common Divisor (GCD) of two integers A and B is the largest integer that divides both A and B.

Write A/B in the form A = B\*Q + R in quotient remainder form

GCD(A,B) = GCD(B,R). Repeat this till either of the numbers is 0, the leftover number is the GCD.

## QuickSort:

Base case – Empty array or an array with one element , as they need not be sorted.  
1. Find Pivot  
Pick an element from array – Pivot  
2. Partitioning

3. Call quicksort recursively on the two sub-arrays.  
quicksort(less) + [pivot] + quicksort(greater)

# MergeSort:

Divide at the mid point until you cannot divide further

Recursively call the Mergesort function on the divided arrays till the array start = end.

# Hash:

Key, value pairs. hash(key) = uniquely identifies the address of value.  
a data structure that lets you express relationships:

Use cases:

* Modelling relationships from one thing to another thing
* Filtering out duplicates
* Caching/memorizing data instead of making your server do work

How to avoid collisions:

• A low load factor – no.of items in hashtable/total no.of slots  
• A good hash function - good hash function distributes values in the array evenly.

# Graph Algorithms:

BFS is graph algo

1. Breadth-first search allows you to find the shortest distance between two things.

To solve a shortest-path problem is called *breadth-first search*.

To figure out how to get from Twin Peaks to the Golden Gate Bridge, there are two steps:

1. Model the problem as a graph.
2. Solve the problem using breadth-first search.

A graph models a set of connections. With Node and Edges.  
BFS It can help answer two types of questions:

• Question type 1: Is there a path from node A to node B?  
• Question type 2: What is the shortest path from node A to node B?

Always first-degree connections are added to the search list before second-degree connections.

BFS Algo:  
First add all the first degree connections to the queue.

Then run the is\_the\_target function on each item of the queue

If the item satisfies the is\_the\_target function well and good, If not add the item’s connections to the queue, at this time, you need not worry as the queue if FIFO, so only the first degree connections will be checked first, though we are adding the second degree connections to the queue.

Never worry about how you will implement the solution in Code. Always think of the approach and the edge cases first, before thinking of implementation.

A graph becomes a tree where no edges ever point back.   
Weighted graph is weighted edges.

Undirected graph – both nodes point to each other which indeed forms a circle.

Dijkstra’s algorithm only works with *directed acyclic graphs*

Shortest path – BFS

Fastest path – Dijkstra’s algorithm

Dijkstra’s algorithm:

Here comes weighted graphs, graphs whose edges have weight.  
Keep on calculating the shortest path from start node to other Nodes. The reference is start node. From start node, keep calculating the shortest path to the other Nodes including Finish. Here the Other Nodes imply the neighbours of the cheapest node.

*Look at the cheapest node on your graph. There is no cheaper way to get to this node!*

If you put a node on the table , it means there is no other cheaper way to get to that node.

Implementation:

1. Represent graph as key value pairs, mapping each node to its neighbours
2. Costs Map – Start node is the reference, calculate the cost of each node from start and map it as value , From start to all nodes of the graph, including finish, this is what the hash map would have.
3. Hash table for parents
4. Processed nodes list, because you don’t want to process a node twice.

This runs in a while loop till there are nodes left to be processed:  
 BFS is same as level order traversal

Greedy Algorithms:

Pick the solution that works for now, locally. Then make it a global solution.

Greedy algorithms are not perfect solutions, but they are good enough. They are approximation algorithms.

Every possible subset that can be formed with n items is 2^n.

## Depth First Search:

To represent graph, pick a default dict, with keys as the nodes with edges, values as the edges nodes.

Start from a Node, Mark that Node as Visited and proceed to the next unmarked node.

Do the same for the next node, till there is no adjacent unmarked node.

BFS needs Queue, DFS needs Stack

# Dynamic Programming:

Dynamic programming starts by solving subproblems and builds up to solving the big problem.

Greedy vs Dynamic:  
Greedy just solved the problem now and then moves on to the next set whereas dynamic programming builds on the solution by looking back to the previous set of solution.

Dynamic Programming – Uses a grid  
Value at cell[i][j] = max of previous value at cell[i-1][j]   
 vs

Value of current item + value of items that can fit in remaining space.

The second value need not be calculated again, you can take the reference from the subproblem which is already solved.

### Tip – two pointer approach or Sliding window approach:

If at any point, you see nested for loops, think of two pointer approach.  
You can use two pointer approach, if the mention consecutive, contiguous items.

# Data Structures:

DataStructures --> Structures to store data

Two types: Linear DataStructures, Non-linear DataStructures

Linear DataStructures:

Have a logical start and logical end

You should know when to use which data structure.  
To find duplicates or to know if an item is already visited, use hash  
To reverse strings – Use Stack

## Arrays:

### Tip:

XOR of an element with 0 returns the element

XOR of an element with itself returns 0

XOR symbol 🡪 ^

### Tip:

First preference is to remove for loop, if that is not possible, try to minimise the if condition checks.

## Linked Lists:

Nodes 🡪 [Data+Pointer]

Linked list will start off with Head node.

Ease of insertion and dynamic size, but difficulty with access.

## Strings:

All possible permutations of a string – n!

## Heap:

Heap is used in Priority queue, heapsort etc.

How to represent heap as an array:

Root of the heap - Arr[0].

If node is at Arr[i], left child at Arr[2\*i + 1] and right child will be at Arr[2\*i+2].

If any node at Arr[i], its parent node will be at Arr[(i-1)/2].

In heapify we always swap the parent with child if it is not following heap property.

## Graph:

Graph can be represented in Adjacency Matrix and Adjacency list.

Adjacency matrix is V\*V array, where V is the number of vertices.

If there is a connection between vertex 1 and vertex 2, the corresponding item in the matrix is marked as 1, else it is 0. Finding a connection is easy, but lot of space is wasted.

Adjacency list – linked list type of representation, where we show all the connected nodes pointed by ->. It saves space, but to know if a node is connected to another takes time as we need to traverse through all the nodes to reach to our node.

## Binary Search Tree:

Binary search tree conditions:

1. Left is less than the node which is less than the right node
2. No duplicate nodes
3. At most 2 sub-nodes like binary tree

Easy for search

## AVL Tree:

Binary search tree with balance factor as 1 or 0.

Balance factor is height of left subtree – height of right subtree.

Time complexity of BST operations(search, insert, delete etc) is O(h) where h is the height of the tree. If the tree is properly distributed the height of the BST tree would be O(logn). If the binary search tree is skewed towards one side, the height could be n and time complexity increases to O(n). To ensure height remains log(n) AVL tree is useful.

Insertion in tree:

A new node is always inserted as a leaf node. Search till you reach the leaf node, once is it is found, add it accordingly. To balance the tree, it is either right rotation or left-right rotation.

For AVL trees, search is faster and insertion and deletions are slower. If we need frequents insertions, red black trees are preferred.

## Red-Black Tree:

Tree whose Nodes have colour Red/Black.

Root and leaf nodes – Black.

No two adjacent nodes with red nodes, hence Red node children are Black.

Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes

## B-Tree:

Difference between binary tree and B-Tree is that, binary tree can have at most 2 nodes, whereas B-Tree can have at most M subnodes, where M is the order of the tree.

In binary tree, records are stored in RAM for faster access. In B-Tree, they are stored in the disk, and the time taken is reduced by limiting the height of the tree by increasing the branches.

B-Tree is also balanced sort tree where nodes are sorted like binary search tree, in In-Order traversal.

## Trie:

Trie is memory efficient than hashmap for dictionary lookup.

# Topological Sort:

First vertex is always a vertex with in-degree as 0, i.e., no incoming edges.

Only works with Directed acyclic Graph.

Only condition is that if an edge is directed from u🡪 v , u should always come before v. For this we use DFS but instead of normal stack which DFS uses, we use a temporary stack, wherein, before popping an element from stack, we check if all the adjacent vertices of an element are already present in the stack.

# Practical Exercises:

### LRU cache

### ZigZag tree traversal:

# Design:

## Scrabble game:

100 letter tiles.

To know which player goes first, the players would randomly pick a letter from the bag and whoever is close to ‘A’ would play first.

Then each player picks 7 letter tiles. The first player’s word sum would be doubled. If the sum of words of the first player is 10, it becomes 20.

If you dispute other player’s word then you lose your turn.

If you want to exchange tiles from the bag, you can, but you lose your turn.

Bingo – When you use all seven letters to form a word. 50 points bonus.

# Tennis tournament:

In a tennis tournament of N players every player plays with every other player.

The following condition always hold-

If player P1 has won the match with P2 and player P2 has won from P3, then Player P1 has also defeated P3.

Find winner of tournament in O(N) time and O(1) space. Find rank of players in O(NlogN) time.

This can be solved using max heap property. Create a max heap and add players into heap. The winner will always be at the root and all the other players will be after that in the order of their defeat. Creating a max heap from array takes 4n run time i.e. O(n). And heap sort is in place sorting algo so O(1) space. To find rank of players we just have to sort this heap which takes O(n.lg n) time.

## Heapify an Array:

Array of n elements.

Last non-leaf node is present at (n/2) -1 th index.

Heapifying single node takes o(logn) complexity, heapify the entire array of n nodes takes nO(logn) complexity.