Basics of Data Structures

**HOW TO APPROACH A DSA PROBLEM:**

1. Understand problem. Note the input data and output data’s properties.
2. Figure out the data structure required
   * 1. **Array**
     2. **Hash Map**
     3. **Stack**
     4. **Queue**
     5. **Linked List**
     6. **Tree**
     7. **Graph**
     8. **Heap/ Priority Queue**
3. Find if any general technique is applicable
   * 1. **Two pointer technique**
     2. **Sliding Window technique**
     3. **Recursion**
     4. **Backtracking**
     5. **Dynamic Programming**
     6. **Greedy**
     7. **Divide and conquer**
4. Make algorithm, write it fully in shortest form. Check for side cases
5. Code it up

#### TOPICS

1. Data Structure and Algorithms
2. Data Structure and Algorithmsic complexity
3. Recursion
4. Backtracking
5. Divide and Conquer

GO LEARN RECCURENCE RELN AND RECURRENCE METHOD !

# Data structure

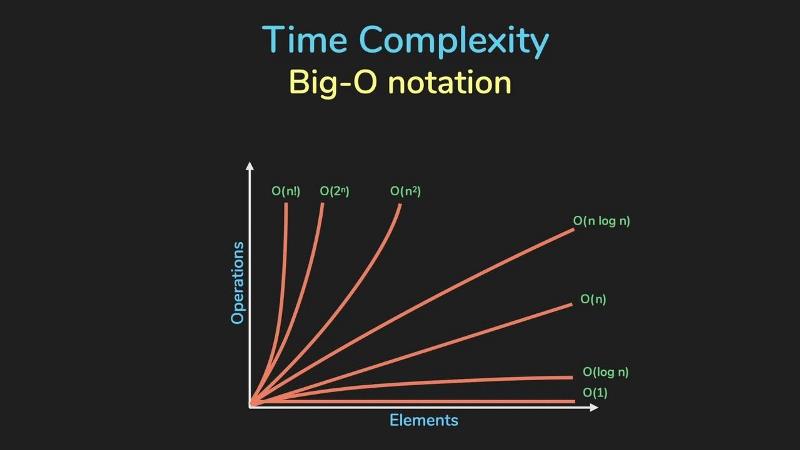
* refers to the structure in which we store data
* is of two types
  + Linear : Array, Linked Lists, Stack, Queue
  + Non-Linear: Trees, Graphs

#### Abstract Data Type (ADT)

An ADT is a high-level description of a data structure that defines a set of operations that can be performed on the data structure and the behavior of those operations. It focuses on what operations can be performed and what their expected behavior is, rather than how those operations are implemented.

# Algorithm analysis

An algorithm is the step-by-step instructions to solve a given problem.

The quality of an algorithm is measured from its **execution time** and **memory requirement**. This can be represented in a polynomial form as shown in figure. This notation is called asymptotic notation. There are three ways to show the complexity of an algorithm:

* **Big O:** Worst case
* **Big Ω:** Best case
* **Big ϴ:** Mixture of the above two. It shows the average case performance.

**Among these, the big O is the most important as it lets us know the upper bound or worst performance an algorithm can perform**.

# Basic Things to know to calculate Complexity

* A loop that iterates 'n' times contributes a factor of 'n' to the complexity.
* Nested loops multiply the complexity by the number of nested levels.
* Such a loop gives a time complexity of O(log n)  
   while ct < 100:  
   print(“Hello”)  
   ct = ct \* 2 # as this variable increases exponentially
* Non looped statements do not contribute to time complexity. Eg: if-else, prints, inputs, etc.

Do many questions from book. Its different everytime. Me skimmed through.

# Recursion

Any function that calls itself is said to be recursive. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. Recursive code tend to be shorter and elegant. But every recursive step requires extra memory (every iterative step does not) and hence recursive solutions tend to be less efficient than its iterative counterpart.

**Basic Format:** def func():  
 if base condition:  
 return value  
 else:  
 return func(changed parameter)

# Backtracking

Backtracking is a more refined and optimized version of **brute force** that intelligently explores the search space, avoiding unnecessary work whenever possible. This is done by generating a **state space trees** and searching for a solution via **Depth First Search**. Backtracking speeds the exhaustive search by cutting tree branches selectively (pruning).

# Divide and conquer

Take a (big) problem and divide it into smaller parts. Now solve these parts. This is **divide**. Now take the output from these smaller problems and combine them to get the final output. This is the **conquer** step.

Basic Format

def DAC(P):

if P is a small problem:

Execute(P)

else:

Divide P into P1,P2 ... Pk

Apply DAC(P1), DAC(P2) ... DAC(Pk)

Combine [DAC(P1), DAC(P2) ... DAC(Pk)]