Differentiate b/w data type and data structure:

Data type:

- Refers to a kind of value a variable can hold and the operations that can be performed on it.
- Build in or primitive type in programming languages like integers, floats, chars and bools.
- Example in python, ints, floats, str, and bool are data type.
- Purpose to store individuals values with specific properties.

Data structure:

- A way of organizing and storing multiple data elements to perform complex operation efficiently.
- Data structures can built using data type and essential for creating diff algorithms.
- Example include : array, linkedlist, stacks, queues, trees and graphs.
- To perform operation like insertion, deletion, and searching efficiently.

Q2 Operation or Array, linked list, Stack Queue (with Pros and cons).

(1) Array.

- Operation:

Acess, Insertion, Peletion, Transversal, Searching.

- pros:
  - constant time acess to elements using an index.
  - simple and efficient for sequential data storage.
- Cons:
  - fixed size if (static) makes resizing difficult
  - Insertion/deletion can be slow if elements need to shift positions.

(2) Linked List:

- operation:

Insertion/deletion, transversal, searching.

- Pros:
- Dynamic size allows easy add/remove of elements.
- Efficient memory usage for sparse data.
- Cons:
- Slower acess to elements (no direct index acess)
- Require extra memory for storing Pointers

③ Stack:

- **Operations:**

  Push (insert), pop (remove), Peek (retrieve top element).

- **Pros:**
  - Lifo (Last In first out) structure is useful for function calls, backtracking and undo operations.
  - Simple to implement.
- **Cons:**
  - Limited access to only the top element.
  - fixed size (if implemented using an array) can lead to stack overflow.

④ Queue:

- **operation:**

  Enqueue (insert) Dequeue (remove), Peek (view front element).

- **pros:**
  - fifo (first In first out) structure is useful for scheduling tasks and buffering.
  - Efficient in managing order-based operation.
- **Cons:**
  - Limited acess to only the front element.
  - fixed size (if array-based), which can lead to overflow or underflow.

Q3 steps for writing an Efficient algorithm.

① Define problem clearly:
- understand the problem requirement and constraints.
- Define inputs, output, and any edge cases.

② Identify Key Requirement:
- consider the resources required (time and space).
- Decide which requirements (e.g.; speed > memory) to prioritize based on constraints.

③ Design a High-level solution:
- Break down the problem in to smaller sub-problems.
- use Pseudocode or flowcharts to outline the solution.

④ Choose the Right Data structures:
- Select data structure that optimize performance.
- Align data structures with operations needed.

⑤ Optimize for Time space complexity:
- Aim to reduce Big-O time and space complexity.
- Implement shortcuts or heuristics if applicable.