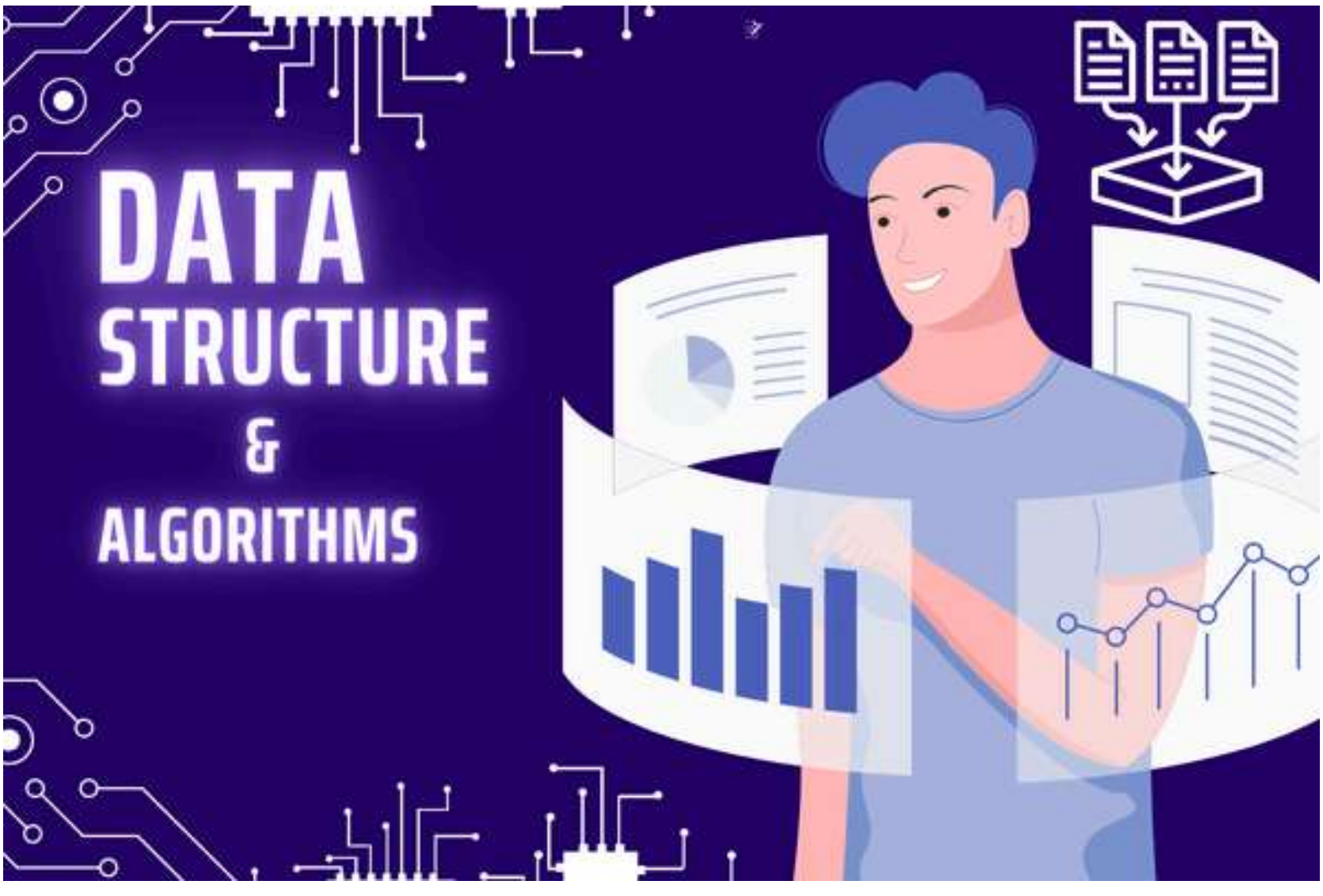
Introduction to Data Structure & Algorithm in Python

Ashraful Islam Mahi

Full Stack Python Developer | ML & Robotics Enthusiast | Founder & CEO, PytronLAB



What is ***Data Structure*** ?

- Data Structures are different ways of organizing data on your computer, that can be used effectively.

Look at these pictures, suppose all the people want to buy ticket for travelling. But all the people are standing randomly before the ticket counter in the left image. on the other hand all the people are organized in a line in he right image. Now tell me, in which case the ticket selling process will be more efficient?

Ofcourse it will be efficient when the people are in a orderly fashion shown in the right image. In the same way it is mandatory to follow some data structures while doing operationons the data.

What is ***Algorithm*** ?

- It can be defined as **"Set of steps to accomplish a task"**.
  Here is a real life example:

Why DSA is Important?

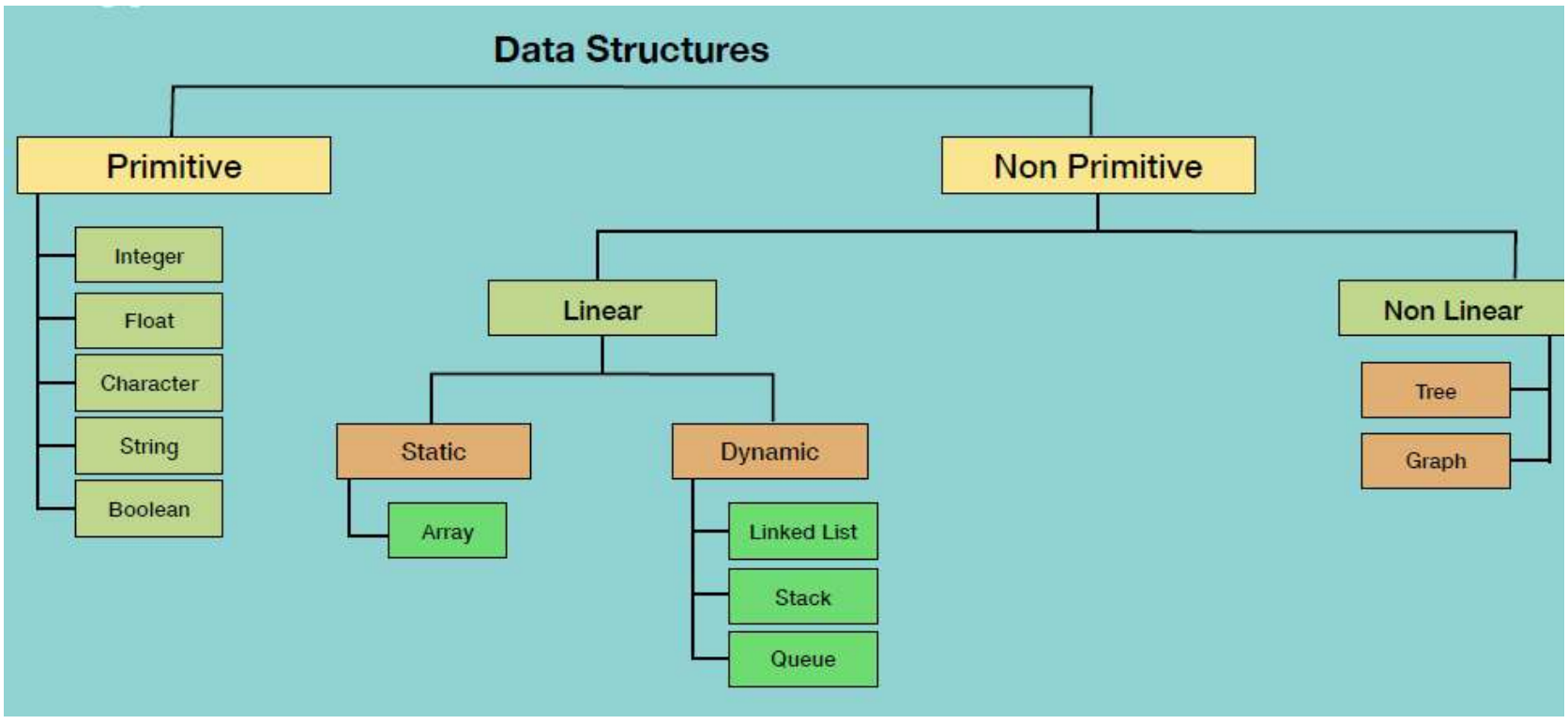
Look at this picture:

Would you like to find your desired book from this library? Hello no!! you won't, Because the books are not properly organized. Similarly it is so much important for us to implement proper *Data Structure & Alhorithm* for memory management and prevent data leakage.

Real life Applications of DS:

- Finding shortest path on a map.
- Used to arrange solar panels on the International Space Station.
- Heavily used in Data processing.

Types of Data Structures & Algorithms:



Types of Algorithms:

1. Simple recursive algorithms:
    - a way of solving a problem by having a function calling itself.
2. Divide and conquer algorithms:
    - Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively.
    - Combine the solutions to the subproblems into a solution to the original problem.
3. Dynamic programming algorithms:
    - They work based on memoization.
    - To find the best solution.
4. Greedy algorithms:
    - We take the best we can without worrying about future consequences.
    - We hope that by choosing a local optimum solution at each step, we will end up at a global optimum solution.
5. Brute force algorithms:
    - It simply tries all possibilities until a satisfactory solution is found.
6. Randomized algorithms:
    - Use a random number at least once during the computation to make a decision.

Time & Space Complexity

What is *Time* Complexity?

- A way of showing how the run time of a funtion changes as the size of input changes.

- It doesn't count how much time an algorithm takes to run , rather it counts the number of operation within a code.
- The developers preper to keep the time complexity as much constant as possible.

What is *Space* Complexity?

- Space complexity measures how much extra memory (RAM/storage) your algorithm uses as the input size grows.
- It helps optimize memory usage and avoid crashing due to memory overflow.

Real-Life Analogy:

- Time complexity = How long it takes to read a book 📖
- Space complexity = How many pages the book takes 📄

Big O:

What is *Big O* notation?

- Big O is the language and metric we use to describe the efficiency of algorithms.
- Measures *Time* & *Space* Complexity.
- It is a way of mathematically figuring out which code is better among many other codes of same purpose.
- Predict performance for large inputs.

Common *Big O* Notations:

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

1. ***O(1) – Constant Time:*** Always takes same time.

In [ ]:
```python
def get_first_item(arr):
    return arr[0]
```

2. ***O(n) - Linear Time:*** Processes each element once.

In [ ]:
```python
def print_items(arr):
    for item in arr:
        print(item)
```

3. ***O(log n) – Logarithmic Time:*** Reduces the problem size each time — like binary search.

In [ ]:
```python
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return True
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return False
```

4. ***O(n^2) – Quadratic Time:*** Nested loops over input — example: Bubble Sort.

In [ ]:
```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

5. ***O(2^n) – Exponential Time:*** Algorithms that double the problem size at each step — e.g., recursive Fibonacci.

In [ ]:
```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Big-O Complexity Chart:

This chart indicates the performance of the Big-O notations. Here it is: