Data Structures & Algorithms with Python

BEAUTY AND JOY OF COMPUTING

- BJC GROUP

Going one level deeper

- In the previous lectures, we covered how to write code and applying abstraction to make program simpler.
- Python already applies "algorithms" to its data structures like lists, and other functions reducing the need for having to code mechanisms such as sorting.
- Though this is helpful, uncovering what it does under the hood, gives us more options to improve our programs, making them faster and efficient.

Data structures and Algorithms

- Data structures are methods to organize and store data efficiently.
- Examples:
  - Linear: Arrays, Linked Lists, Stacks, Queues.
  - Non-Linear: Trees, Graphs, Heaps.

- Algorithms are step-by-step instructions to solve problems.
- Examples:
  - Sorting: Bubble Sort, Merge Sort.
  - Searching: Binary Search, Depth-First Search (DFS).

Example algorithm

- Imagine you are in 1980s when they used phone books.
- The goal is to find John Mayer's phone number of the 1000s of numbers **ordered in an ascending** manner.
- First method you may think of intuitively is searching number by number from the first page, however, this will take forever to get the Js.
- Next you may think of opening the middle page, comparing the letter you find with John.
  - if it's greater than John, then you might, cut the first half of the book, and start from the middle page of the second half of the book.
  - Otherwise cut the second half and check the middle page of the first half.
  - Repeat the process until you find John.
- We apply the same intuition to writing algorithms.

Complexity Analysis

- Complexity analysis is the measure of the performance of an algorithm.
- We consider the growth rate as the size of input (n) increases.
- Two types:
  - Time Complexity: Execution time as input size grows.
  - Space Complexity: Memory usage as input size grows.

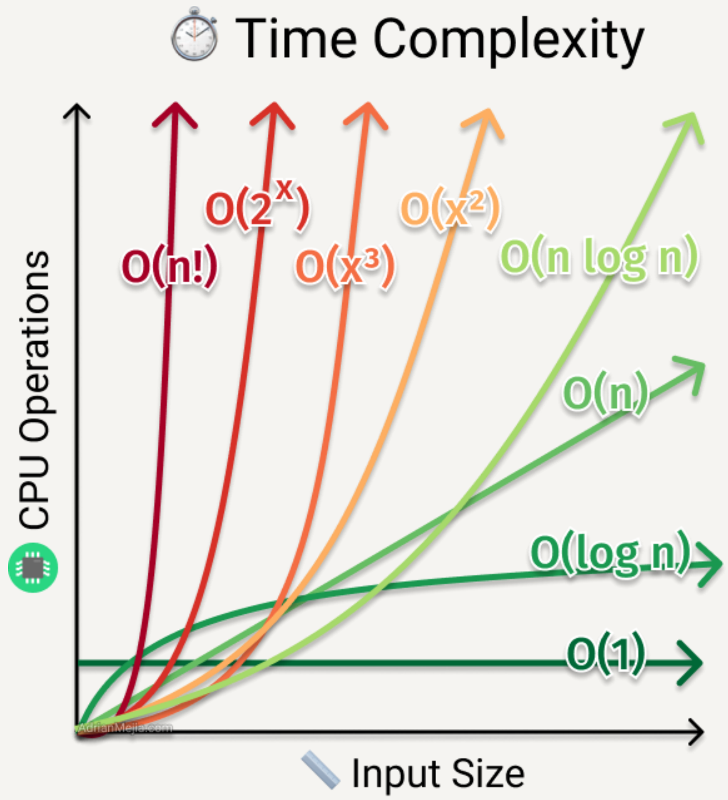
- Asymptotic Notation
  - **Big-O (O)**: Worst-case complexity.
  - **Omega (Ω)**: Best-case complexity.
  - **Theta (Θ)**: Average-case complexity.

Complexity Analysis

- In complexity analysis tradeoffs is usually considered. To get faster programs, there might be a need to sacrifice space

- The graph shows the Big-O (Worst case **time** complexity) as for different algorithms.

- We want to get as close as to O(1) [constant time] if possible.



Complexity Analysis

- Common time complexities
  - **$O(1)$**: Constant time (e.g., accessing an array element).
  - **$O(\log n)$**: Logarithmic (e.g., Binary Search).
  - **$O(n)$**: Linear (e.g., Linear Search).
  - **$O(n \log n)$**: Log-linear (e.g., Merge Sort).
  - **$O(n^2)$**: Quadratic (e.g., Bubble Sort).
  - **$O(2^n)$**: Exponential (e.g., Recursive Fibonacci).

Arrays

- Arrays are fixed-size, contiguous memory storage for elements of the same type.
- List (Python) : Dynamic, ordered collection of elements of any type.
- Features:
  - Elements are indexed starting from `0`.
  - Arrays require manual size declaration; Python lists adjust dynamically.
  - Allow duplicate elements.

Arrays

- Common Operations:
  - Access an element: `array[i]` or `list[i]`.

- Use Cases:
  - Sequential data storage.
  - Easy access via indices

Linked Lists

Linked lists are a dynamic, linear data structure where elements (nodes) are linked by pointers.

- Types of Linked Lists
  - Singly: Each node points to the next node.
  - Doubly: Nodes have pointers to both next and previous nodes.
  - Circular: Last node points back to the first node.
- Features:
  - Dynamic size (no fixed size like arrays).
  - Sequential access (no direct indexing).

Linked Lists

- Common Operations:
  - Add Node: At head, tail, or specific position.
  - Remove Node: By value or position.
  - Traverse: Visit each node sequentially.
- Use Cases:
  - Dynamic memory allocation.
  - Implementing stacks, queues, or adjacency lists in graphs.

Stacks

Stacks are a LIFO (Last In, First Out) data structure.

- Operations:
  - Push: Add element to the top.
  - Pop: Remove top element.
  - Peek: View top element without removing.
- Use Cases:
  - Undo functionality.
  - Expression evaluation (e.g., postfix notation).

Queues

Queues are a FIFO (First In, First Out) data structure.

- Operations:
  - Enqueue: Add element to the rear.
  - Dequeue: Remove element from the front.
  - Peek: View front element without removing.
- Types:
  - Simple Queue, Circular Queue, Priority Queue.
- Use Cases:
  - Scheduling tasks.
  - Handling requests in order (e.g., printer queue).

Trees

- Trees are a non-linear hierarchical data structure with nodes.
- Key Terms:
  - Root, Parent, Child, Leaf, Sibling.
- Types:
  - Binary Tree: Each node has ≤2 children.
  - Binary Search Tree (BST): Left < Root < Right.
  - AVL/Red-Black Tree: Balanced trees.
- Common Traversals:
  - Preorder, Inorder, Postorder (DFS), Level Order (BFS).
- Use Cases:
  - File systems, Expression evaluation, Search operations.

Graphs

xGraphs are non-linear structure of nodes (vertices) connected by edges.

- Key Features:
  - Directed or Undirected.
  - Weighted or Unweighted.
  - Representation:
  - Adjacency Matrix.
  - Adjacency List.
- Common Algorithms:
  - DFS, BFS, Dijkstra’s Algorithm, Kruskal’s Algorithm.
- Use Cases:
  - Network routing, Social networks, Web crawling.

Sorting Algorithms

- Sorting in programming is organizing elements in a specific order (ascending or descending).
- Helps improve efficiency in data searching and organization.
- Types of Sorting Algorithms:
    - Simple sorting
    - Efficient sorting
    - Specialized sorting

Simple sorting

- Bubble Sort: Compare adjacent elements; swap if out of order.
  - Complexity: $O(n^2)$.
- Selection Sort: Find the smallest element and place it in sorted order.
  - Complexity: $O(n^2)$.
- Insertion Sort: Build the sorted list one element at a time.
  - Complexity: $O(n^2)$.

Efficient Sorting

- Merge Sort: Divide-and-conquer algorithm; split, sort, and merge.
  - Complexity: $O(n \log n)$.
- Quick Sort: Partition array around a pivot, recursively sort partitions.
  - Complexity: $O(n \log n)$ (average), $O(n^2)$ (worst case).
- Heap Sort: Use a binary heap to sort elements.
  - Complexity: $O(n \log n)$.

Specialized sorting

- Counting Sort: Count occurrences of elements; good for integers.
  - Complexity: $O(n + k)$.
- Radix Sort: Process digits one place at a time; uses counting sort as a subroutine.
  - Complexity: $O(nk)$.

Search Algorithms

Search Algoriths are used to find the position or presence of an element in a dataset.

Essential for data retrieval and organization.

Types of Search Algorithms:

- Linear Search
  - Traverse the dataset sequentially.
  - Complexity: O(n).
  - Use Case: Small or unsorted datasets.

Search Algorithms (continued)

- Binary Search
  - Divide-and-conquer; requires sorted data.
  - Check the middle element; eliminate half of the dataset.
  - Complexity: $O(\log n)$.
  - Use Case: Large, sorted datasets.

- Hashing
  - Use hash functions to directly access elements.
  - Complexity: $O(1)$ (average), $O(n)$ (worst case).
  - Use Case: Fast lookups in hash tables.

Recursion

- Recursion is a method where a function calls itself to solve a smaller instance of the problem.
- Idea: Break a problem into smaller subproblems.

- Components of Recursion
  - Base Case: The stopping condition to prevent infinite recursion.

```
if n == 0:
    return 1
```

  - Recursive Case: The function calls itself to solve the smaller problem.

```
return n * factorial(n - 1)
```

Recursion(continued)

- Benefits of recursion:
  - Simplifies code for problems that have repetitive substructures (e.g., trees, fractals).
  - Natural fit for divide-and-conquer algorithms.

```python
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:       # Recursive case
        return n * factorial(n - 1)
```

Recursive function implementation of n-factorial

Project challenge

- Implement a rucursive function for fibonacci sequence!
- Fibonacci sequences is a series of numbers starting with 0 and 1 and the next digits are found by adding up the two numbers before it.
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- 2 is found by adding the two numbers before it (1 + 1)
- 8 is found by adding the two numbers before it (3+5) and so on..
- Tips: Remember to add the base case, something like if n == 0 or n ==1