

Data Structures & Algorithm



The Building Blocks of Software Engineering

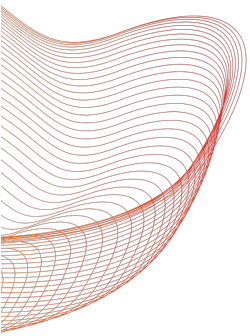


Introduction

Welcome to the exploration of
Data Structures and Algorithms!

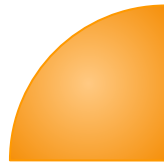
How are these concepts central to
Software Engineering?

Together, we will delve into the
role they play in crafting efficient
and effective software.





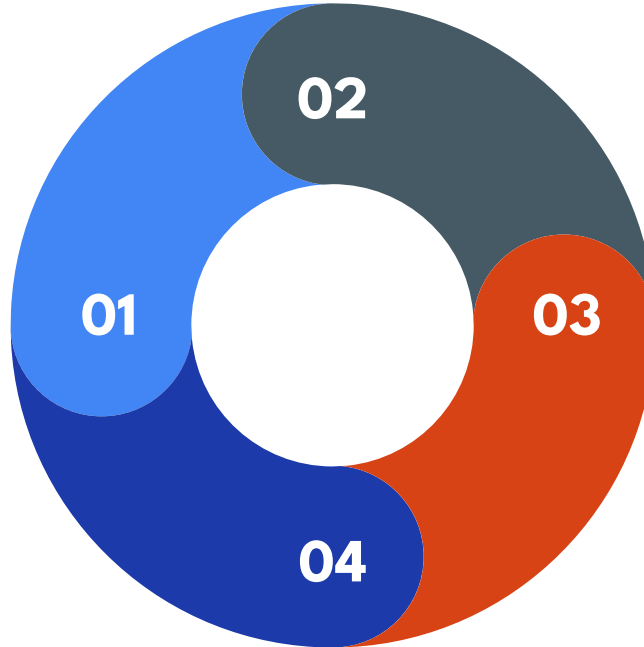
What are Data Structures?

- Data structures are a specific means of organizing and storing data in a computer.
 - These structures enable efficient execution of complex tasks.
 - Data structures bring structure to raw data, enable access to data items, and allow various operations, such as insertions, deletions, and modifications.
- 

The Importance of Data Structures

The right choice of data structure can enhance algorithm efficiency.

Data structures help in organizing and storing data.

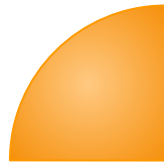


They serve as the basis for abstract data types (ADT).

The use of data structures can lead to efficient software design and development.



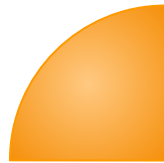
Understanding Big O Notation

- Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
 - In computer science, it's used to classify algorithms by how their run time or space requirements grow as the input size grows.
 - $O(1)$ denotes constant time complexity, $O(N)$ represents linear time complexity, $O(N^2)$ signifies quadratic time complexity, $O(\log N)$ stands for logarithmic time complexity, and $O(N \log N)$ denotes linearithmic time complexity.
- 



Understanding Big O notation (Cont'd)

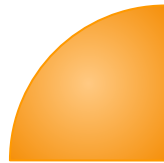
$O(1)$ - Constant Time Complexity: An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Example: Array access operation.





Understanding Big O notation (Cont'd)

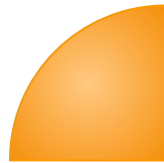
$O(N)$ - Linear Time Complexity: An algorithm has linear time complexity if its running time increases linearly with the input size. Example: Unsorted array search.





Understanding Big O notation (Cont'd)

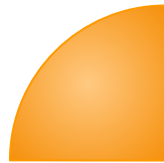
$O(N^2)$ - Quadratic Time Complexity: An algorithm is quadratic time if its running time is proportional to the square of the input size. This is common with algorithms that involve nested iterations over the input data. Example: Bubble Sort.





Understanding Big O notation (Cont'd)

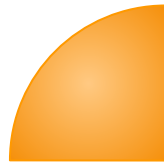
$O(\log N)$ - Logarithmic Time Complexity: An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step. Example: Binary Search.



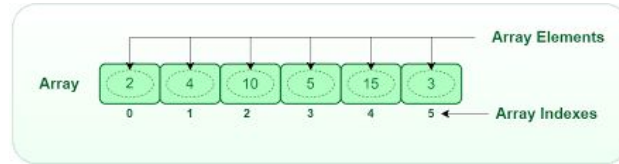


Understanding Big O notation (Cont'd)

$O(N \log N)$ - Linearithmic Time Complexity: An algorithm is said to have linearithmic time complexity if its running time is proportional to $n \log n$. These algorithms often involve a divide and conquer strategy where the input is divided in each step. Example: Merge Sort.

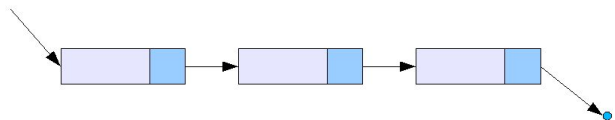


Arrays



- An array is a collection of elements, each identified by an array index or key.
- Arrays store a fixed-size sequential collection of elements of the same type.
- Operations:
 - **Access:** Retrieve an element at a given index. This operation takes $O(1)$ time.
 - **Search:** Check for an element using given index or by traversing. This operation takes $O(n)$ time.
 - **Insertion:** Add an element at a given index. This operation takes $O(n)$ time.
 - **Deletion:** Remove an element at a given index. This operation also takes $O(n)$ time."
- **Applications:** Databases, CPU scheduling, symbol table implementation, etc.

Linked Lists



- A Linked List is a linear data structure where each element is a separate object.
- Elements aren't stored at contiguous locations; instead, they're linked using pointers.
- Operations:

Insertion: To add a node, we just need to adjust the node pointers. This operation is $O(1)$ if done at the head and $O(n)$ if done at the tail.

Deletion: Similar to insertion, we rearrange pointers to remove a node. It's $O(1)$ if performed at the head and $O(n)$ if performed at the tail.

Traversal: We start from the head and follow the node links. It takes $O(n)$ time.

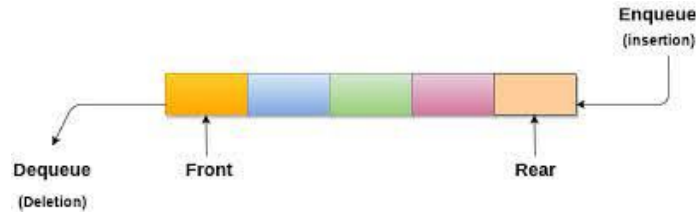
- **Applications:** Dynamic memory allocation, implementing stacks and queues, performing arithmetic operations on long integers, etc.

Stack

- A Stack is a linear data structure following the Last-In-First-Out (LIFO) principle.
- **Operations:**
 - **PUSH:** Adds an element to the top of the stack. This operation takes $O(1)$ time.
 - **POP:** Removes an element from the top of the stack, which also takes $O(1)$ time.
 - **PEEK/Top:** Gets the top data element of the stack without removing it, a constant time operation $O(1)$.
- **Applications:** Expression Evaluation, Syntax Parsing, Backtracking algorithms, etc.



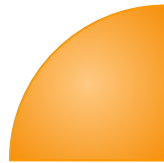
Queue



- A Queue is a linear data structure following the First-In-First-Out (FIFO) principle.
- **Operations:**
 - **ENQUEUE:** Adds an element to the end of the queue, a constant time operation $O(1)$.
 - **DEQUEUE:** Removes an element from the front of the queue, which is also a constant time operation $O(1)$.
 - **PEEK/Front:** Gets the front item from queue without removing it, another $O(1)$ operation.
- **Applications:** CPU Scheduling, Disk Scheduling, IO Buffers, etc.

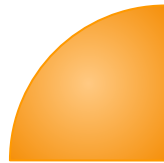


Other Data Structures

- **Trees:** a non-linear, hierarchical data structure with a root value and subtrees of children, represented as a set of linked nodes.
 - **Graphs:** a non-linear data structure consisting of nodes and edges.
 - **Hash tables:** a data structure that implements an associative array abstract data type, a structure that can map keys to values.
- 

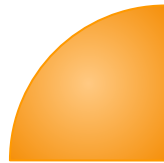


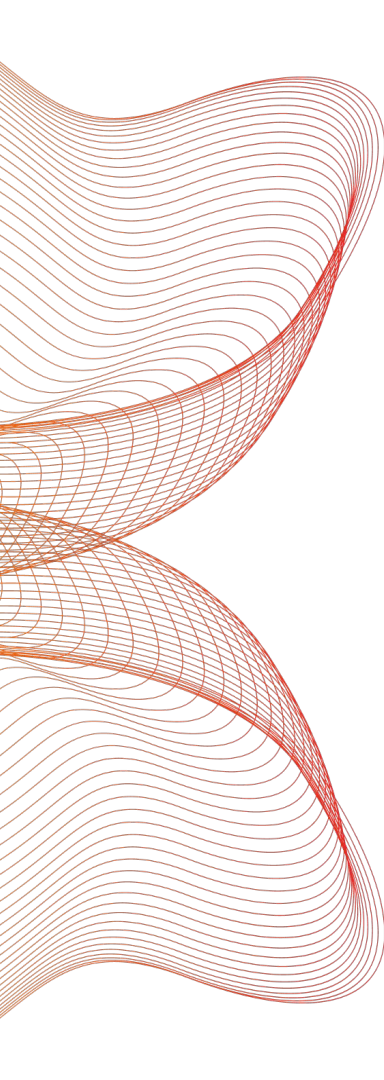

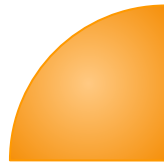
Algorithms

- Algorithms are a finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems.
 - Algorithms have a clear starting and ending point, and a finite number of steps.
- 



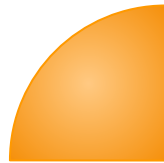
Importance of Algorithms

- Algorithms can solve a problem in the most efficient way.
 - The efficiency of a program can be measured by analyzing the algorithms it uses.
 - Algorithms allow us to understand scalability and the restrictions of an approach.
- 

- 
- 
- **Brute Force Algorithms:** Solve the problem by trying all possible solutions.
 - **Recursive Algorithms:** Solve problems by solving smaller instances of the same problem.
 - **Divide and Conquer Algorithms:** Break a problem into non-overlapping subproblems, solve these subproblems independently, and then combine their solutions to solve the original problem.
 - **Dynamic Programming Algorithms:** Break a problem into overlapping subproblems and build up solutions by using the solutions to smaller subproblems.
 - **Greedy Algorithms:** Make the locally optimal choice at each stage in the hope that these local solutions lead to a global optimum.
 - **Backtracking Algorithms:** Try to solve the problem incrementally, by building a sequence of choices.
- 

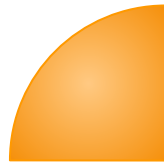


Sorting Algorithms

- **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order.
 - **Selection Sort:** Selects the smallest (or largest) element from the list, places it at the start, and then repeats the process for the remaining list.
 - **Insertion Sort:** Builds the final sorted array one item at a time.
 - **Merge Sort:** Divides the unsorted list into n sublists, then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.
 - **Quick Sort:** Uses the divide-and-conquer strategy to sort the list items.
- 

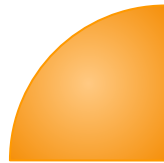


Algorithm Design Techniques

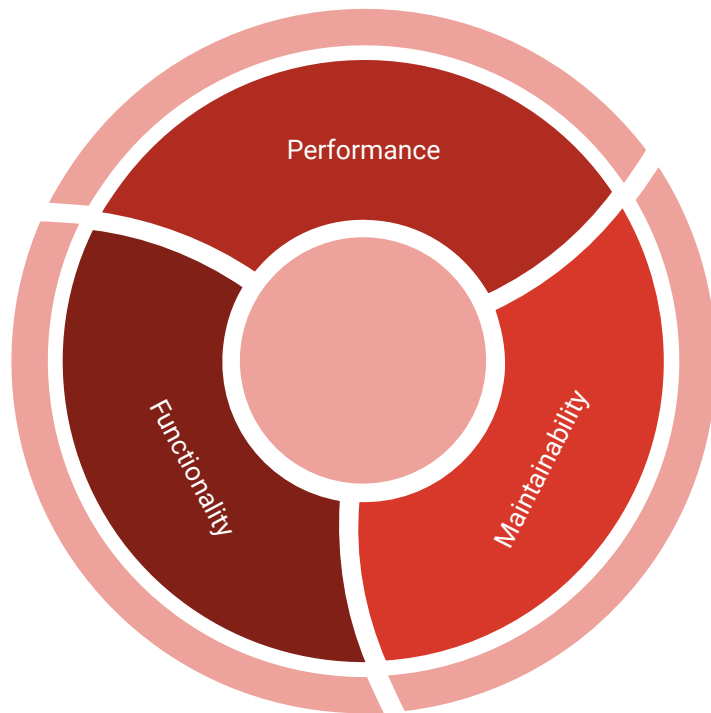
- **Divide and Conquer:** Splits the problem into smaller subproblems of the same type, solves them independently, and combines their solutions to solve the original problem.
 - **Dynamic Programming:** Solves complex problems by breaking them down into simpler overlapping subproblems and reusing solutions from smaller solved problems.
 - **Greedy Technique:** Solves problems by making the most optimal choice at each step.
 - **Backtracking:** Solves problems incrementally by trying out sequences of choices and undoing them if they do not lead to a solution.
- 



Characteristics of a Good Algorithm

- **Unambiguity:** Each of its steps (or phases) is clear and leads to only one meaning.
 - **Well-Defined Inputs:** The algorithm should clearly define which inputs it's working with.
 - **Well-Defined Outputs:** It should be clear what the algorithm will output after processing the given inputs.
 - **Finiteness:** The algorithm must terminate after a finite number of steps.
 - **Feasibility:** It should be simple, generic, and practical so that it can be executed with the available resources.
 - **Independent:** An algorithm should have step-by-step directions that are independently of any programming language.
- 

Striking a balance





More Resources

<https://www.hackerrank.com/>

<https://leetcode.com/>

<https://www.freecodecamp.org/>





Q & A

