**Data Structures & Algorithms (DSA) Documentation for Presentation**

**1. Introduction to DSA**

Data Structures and Algorithms (DSA) form the foundation of computer science and software development. Understanding DSA enables developers to write efficient and optimized code for solving complex problems.

**2. Importance of Data Structures in Programming**

* Organize and store data efficiently
* Enhance performance of programs
* Enable scalability of applications
* Help in managing and processing large datasets

**Practical Examples:**

* **Array**: Storing marks of students in a subject



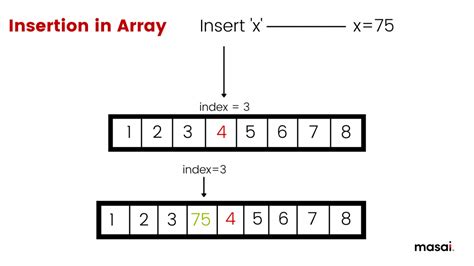
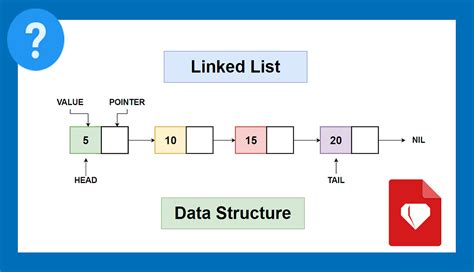
* **Linked List**: Playlist of songs where next song is linked to the current
* **Stack**: Undo feature in text editors



* **Hash Table**: Storing and retrieving user information quickly



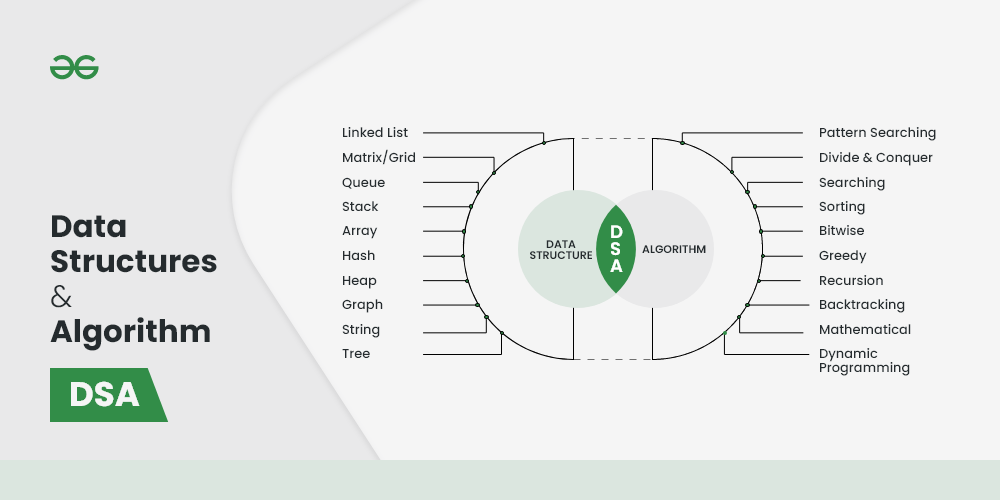
**Visual Reference:**

* 
* 

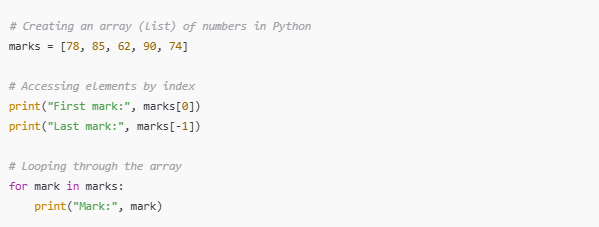
**3. Difference Between Algorithms and Data Structures**

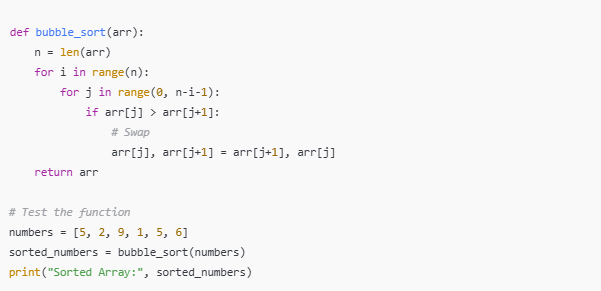
* **Data Structure**: A way to store and organize data (e.g., array, stack, queue, tree)
* **Algorithm**: A set of instructions or steps to perform a task or solve a problem (e.g., sorting, searching)

Data structures hold the data; algorithms operate on the data.



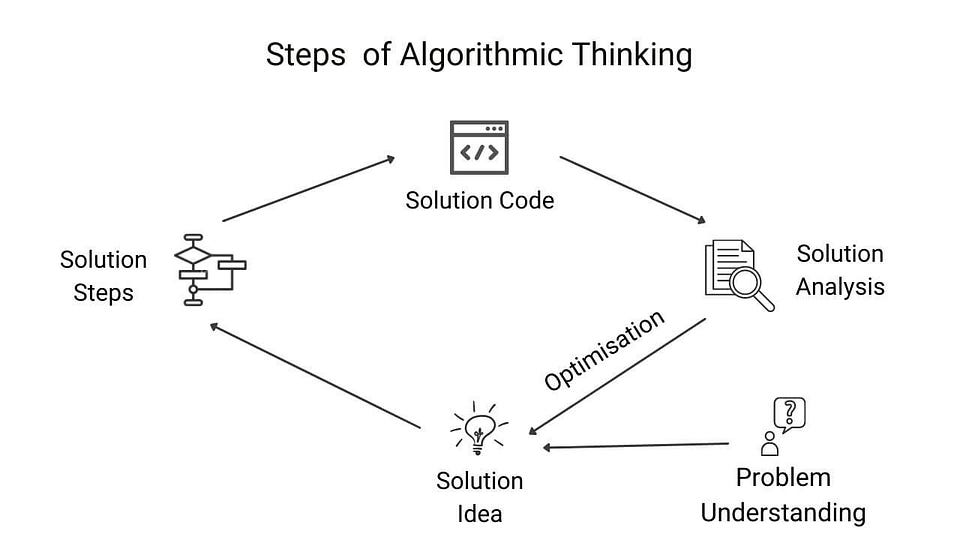
**Practical Example:**

* **Data Structure**: Array of number
* **Algorithm**: Bubble sort algorithm to sort the array



**4. Algorithmic Thinking**

**Definition**: Algorithmic thinking is a step-by-step approach to solving problems using logical thinking and problem decomposition.



**Key Concepts:**

* Identify the problem clearly
* Break down the problem into smaller sub-problems
* Solve each sub-problem step by step
* Combine solutions to solve the main problem

**Example:**

* Problem: Find the largest number in an array
  + Step 1: Initialize max = arr[0]
  + Step 2: Iterate through array with loop
  + Step 3: Compare each element with max
  + Step 4: Update max if a larger element is found

**Code Snippet (Python):**

arr = [10, 5, 20, 8]

max\_val = arr[0]

for num in arr:

if num > max\_val:

max\_val = num

print("Maximum value:", max\_val)

**5. Time Complexity (Big O Notation)**

**Definition**: Big O notation describes the performance or complexity of an algorithm in terms of input size (n).

**Why it's Important**:

* Predicts scalability
* Helps in choosing efficient algorithms

**Examples:**

* **O(1)** – Constant Time: Accessing an array element
* **O(n)** – Linear Time: Iterating through an array
* **O(log n)** – Logarithmic Time: Binary search
* **O(n^2)** – Quadratic Time: Bubble sort

**Practical Example:**

* Linear Search: O(n)

arr = [1, 3, 5, 7, 9]

target = 5

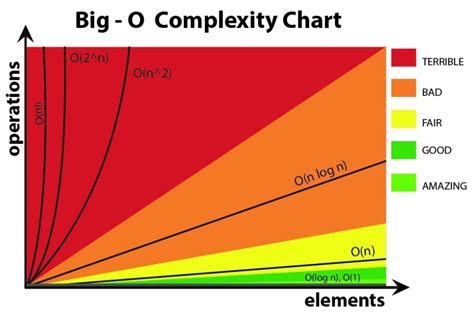
for i in arr:

if i == target:

print("Found")

break

**Visual Reference:**

* 

**6. Worst-case, Best-case, and Average-case Scenarios**

* **Best-case**: The fastest execution scenario
* **Worst-case**: The slowest execution scenario
* **Average-case**: Expected performance for a random input

**Example with Sorting Algorithms:**

* Bubble Sort:
  + Best-case: O(n) (already sorted)
  + Worst-case: O(n^2) (reversed order)
  + Average-case: O(n^2)

**Practical Example:**

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

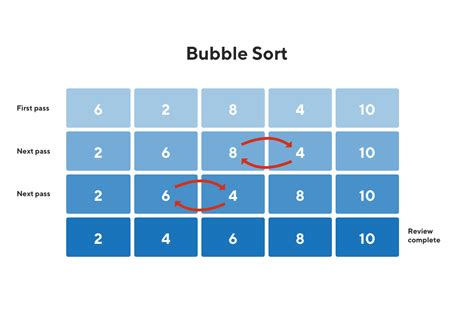
if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

print(bubble\_sort([5, 3, 8, 6]))

**Visual Reference:**

* 

**7. Constant Time (O(1)) Operations**

**Definition**: Operations that take the same amount of time regardless of input size.

**Examples:**

* Accessing a specific index in an array: arr[0]
* Inserting/removing elements from the end of a dynamic array (amortized O(1))
* Hash table lookup

**Practical Use Cases:**

* Retrieving user data from cache:

cache = {"user1": "Alice", "user2": "Bob"}

print(cache["user1"]) # O(1)

* Accessing configuration values from a dictionary:

config = {"theme": "dark", "volume": 75}

print(config["theme"]) # O(1)