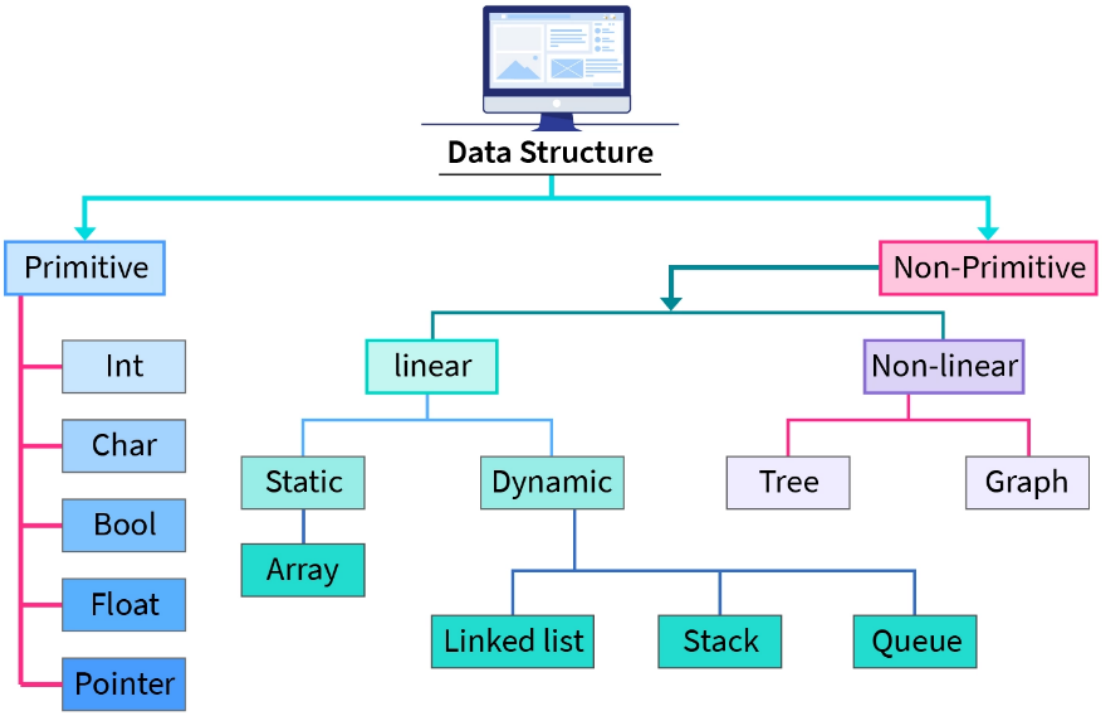
ALGORITHMS

Python algorithms provide a detailed set of instructions by which you can process data for a specific purpose. The most well-known are sorting and graph. Python algorithms are indispensable tools for any software engineer or data scientist. Algorithms are not language-specific and have no standardized rules dictating how they should be written. This means that solutions we've used for decades can be applied as needed to a Python program.



Types of Algorithms in Python:

Python can use a wide variety of algorithms, but some of the most well-known are tree traversal, sorting, search and graph algorithms.

- **Tree traversal** algorithms are designed to visit all nodes of a tree graph, starting from the root and traversing each node according to the instructions laid out. Traversal can occur in order, with the algorithm traversing the tree from node to edge (branches), or from the edges to the root.
- **Sorting algorithms** provide various ways of arranging data in a particular format, with common algorithms including bubble sort, merge sort, insertion sort and shell sort.
- **Searching algorithms** check and retrieve elements from different data structures, with variations including linear search and binary search.
- **Graph algorithms** traverse graphs from their edges in a depth-first (DFS) or breadth-first (BFS) manner.

Writing an Algorithm in Python:

Algorithms written in Python, or any other language are most written in a step-by-step manner that clearly defines the instructions a program needs to run. Though there is no defined standard as to how you should write algorithm, there are basic shared code constructs between languages that we often use to create an algorithm, such as loops and control flow.

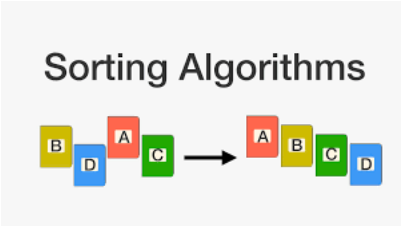
Algorithms are written to solve problems and overcome challenges in development, so ensuring that a problem is well defined is key to writing a solution. Oftentimes, there may be multiple solutions to a given problem and many algorithms may be implemented at once as a way of helping the program find the best solution available.

An algorithm should contain six characteristics:

→ It is unambiguous and has clear steps.
→ The algorithm has zero or more well-defined inputs.
→ It must have one or more defined outputs.
→ The algorithm must terminate after a finite number of steps.
→ It must be feasible and exist using available resources.
→ The algorithm should be written independently of all programming code.

Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.




```
nums = [5, 2, 9, 1, 5, 6]
nums.sort()  # Sorts the list in ascending order
print(nums)
```

Output: [1, 2, 5, 5, 6, 9]

Searching Algorithms

Searching algorithms are techniques used to find specific elements or data within a collection of data, such as a list, array, or database.

→ Linear Search

Linear search is the simplest searching algorithm. It checks each element of the list until the desired element is found or the list ends.

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index of the target element
    return -1  # Return -1 if the element is not found

# Example usage:
arr = [2, 4, 0, 1, 9]
target = 1
result = linear_search(arr, target)
print(result)
```

Output: 3

→ Binary Search

Binary search is a more efficient algorithm but requires the list to be sorted. It repeatedly divides the search interval in half. If the target value is less than the middle element, the search continues in the left half; otherwise, it continues in the right half.

Example:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # Return the index of the target element
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Return -1 if the element is not found

# Example usage:
arr = [1, 3, 5, 7, 9, 11]  # The list must be sorted
target = 7
result = binary_search(arr, target)
print(result)
```

Output: 3

Dijkstra's Algorithm:

To find the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights.

Step-by-step process:

→ dist = {A: 0, B: ∞, C: ∞}

Priority Queue: [(A, 0)]

Step 2: Process:

Extract (A, 0) → Update B: dist[B] = 1, C: dist[C] = 3.

Extract (B, 1) → Update C: dist[C] = 2.

Extract (C, 2)

Result: dist = {A: 0, B: 1, C: 2}

Bellman Ford's algorithm:

To find the shortest path from a single source vertex to all other vertices in a graph, which may have negative weight edges.

Step-by-step process:

→ dist = {A: 0, B: ∞, C: ∞}

Relax all edges V-1 times:

→ Iteration 1:

Relax (A, B, 4): dist[B] = 4

Relax (A, C, -2): dist[C] = -2

Relax (C, B, 3): dist[B] = 1

→ Iteration 2:

Relax (A, B, 4): No update

Relax (A, C, -2): No update

Relax (C, B, 3): No update

→ Step 3: Check for negative-weight cycles:

No negative cycles detected.

Result: dist = {A: 0, B: 1, C: -2}

Graph Traversals:

→ Breadth-First Search (BFS)

Breadth-First Search is an algorithm for traversing or searching tree or graph data structures. It starts at the root node and explores all of the neighbor nodes at the present depth level before moving on to nodes at the next depth level.

Algorithm:

1. Start at the root node (or an arbitrary node in the case of a graph).
2. Mark the current node as visited and add it to a queue.
3. While the queue is not empty:
    - Dequeue a node from the front of the queue.
    - Process the dequeued node.
    - For each unvisited neighbor of the dequeued node:
        - Mark it as visited and enqueue it.
4. Repeat until the queue is empty.

Example:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()

        if node not in visited:
            print(node)  # Or process the node
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)
```

→ Depth-First Search (DFS)

Depth-First Search is an algorithm for traversing or searching tree or graph data structures. It starts at the root node and explores as far as possible along each branch before backtracking.

Algorithm:

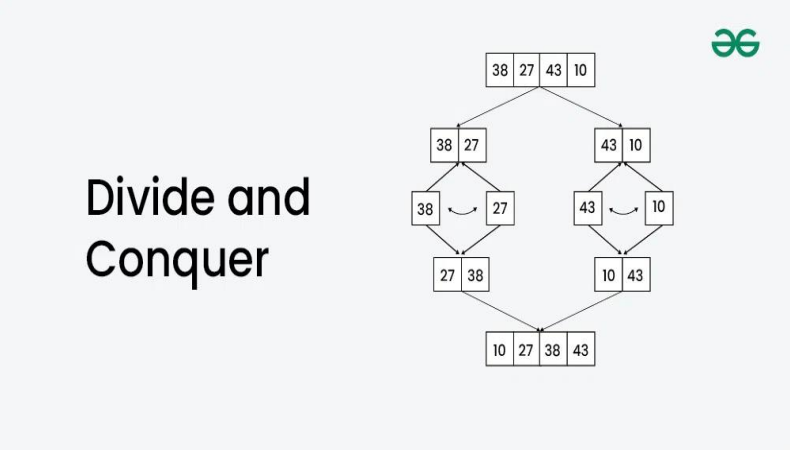
1. Start at the root node (or an arbitrary node in the case of a graph).
2. Mark the current node as visited and print (or process) it.
3. Visit the first unvisited neighbor of the current node.
4. Recursively apply the DFS algorithm to the unvisited neighbor.
5. Backtrack to the previous node when there are no more unvisited neighbors and continue the process.
6. Repeat until all nodes have been visited.

Example:

```
def dfs(graph, node, visited=set()):
    if node not in visited:
        print(node)  # Or process the node
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
```

Divide and Conquer Algorithm

Definition: The Divide and Conquer algorithm is a powerful strategy for solving complex problems. The basic idea is to break down a problem into smaller subproblems that are easier to solve. These subproblems are solved independently, and their solutions are then combined to solve the original problem.



Step-by-Step Process:

The Divide and Conquer approach generally follow three main steps:

Divide:

Break the problem into smaller subproblems. These subproblems should be of the same type as the original problem.

Ideally, the subproblems should be of approximately equal size to reduce the overall complexity.

Conquer:

Solve the subproblems recursively. If the subproblem size is small enough, solve it directly (this is the base case of the recursion).

Combine:

Combine the solutions of the subproblems to form the solution of the original problem.

Example: Merge Sort

To illustrate Divide and Conquer, let's go through the Merge Sort algorithm, a classic example of this approach.

Problem: Sort an Array

→ Step 1: Divide

Divide the unsorted array into two halves until each subarray contains only one element.

Example: Given array [38, 27, 43, 3, 9, 82, 10], divide it as follows:

[38, 27, 43, 3, 9, 82, 10]

[38, 27, 43] and [3, 9, 82, 10]

[38], [27, 43], [3, 9], [82, 10]

[38], [27], [43], [3], [9], [82], [10]

→ Step 2: Conquer

Recursively sort the subarrays.

For arrays of size 1, they are already sorted, so this is the base case.

→ Step 3: Combine

Merge the sorted subarrays to form a single sorted array.

Merge [27] and [43] to get [27, 43].

Merge [3] and [9] to get [3, 9].

Merge [38] and [27, 43] to get [27, 38, 43].

Merge [82] and [10] to get [10, 82].

Finally, merge [27, 38, 43] and [3, 9, 10, 82] to get [3, 9, 10, 27, 38, 43, 82].

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

# Example Usage:
arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print(arr)
```

Output: [3, 9, 10, 27, 38, 43, 82]

Fractional Knapsack Problem

Given a set of items, each with a weight and a value, determine the maximum value of items that can be included in a knapsack of a given capacity. Unlike the 0/1 Knapsack problem, where you can either take the whole item or leave it, the Fractional Knapsack problem allows you to take fractions of an item. This means that if an item can't be fully accommodated, you can take a fraction of it to fill the remaining capacity of the knapsack.

Greedy Approach for Fractional Knapsack

The Fractional Knapsack problem can be solved efficiently using a greedy algorithm. The key idea is to maximize the value-to-weight ratio.

Step-by-Step Process:

1. **Input**:
   - n: Number of items.
   - W: Maximum capacity of the knapsack.
   - items: A list of tuples, where each tuple contains the value and weight of an item, i.e., items = [(value1, weight1), (value2, weight2), ...].
2. **Calculate Value-to-Weight Ratio**:
   - For each item, calculate the ratio of value to weight: ratio = value / weight.
   - Store each item along with its ratio.
3. **Sort Items by Ratio**:
   - Sort the items in descending order based on their value-to-weight ratio.
4. **Pick Items**:
   - Initialize total value as 0 to store the total value of items in the knapsack.
   - For each item in the sorted list:
     - If the entire item can be added to the knapsack without exceeding the capacity, add the full item.
     - If only part of the item can be added, take the fraction that fits, and then break the loop as the knapsack is now full.
5. **Return Result**:
   - The total value at the end is the maximum value that can be obtained with the given knapsack capacity.

Example:

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
        self.ratio = value / weight

def fractional_knapsack(W, items):
```

```
    # Sort items by value-to-weight ratio in descending order
    items.sort(key=lambda item: item.ratio, reverse=True)

    total_value = 0.0

    for item in items:
        if W >= item.weight:
            # Take the whole item
            W -= item.weight
            total_value += item.value
        else:
            # Take the fraction of the remaining capacity
            total_value += item.value * (W / item.weight)
            break

    return total_value

# Example usage
items = [
    Item(60, 10),  # value 60, weight 10
    Item(100, 20),  # value 100, weight 20
    Item(120, 30)  # value 120, weight 30
]

max_capacity = 50
max_value = fractional_knapsack(max_capacity, items)
print(f"Maximum value in the knapsack: {max_value}")
```

Explanation:

- Items: [(value: 60, weight: 10), (value: 100, weight: 20), (value: 120, weight: 30)]
- Knapsack capacity: 50

Step 1: Calculate the value-to-weight ratio:

- Item 1: Ratio = 60 / 10 = 6
- Item 2: Ratio = 100 / 20 = 5
- Item 3: Ratio = 120 / 30 = 4

Step 2: Sort items by the ratio:

- Sorted items: [(60, 10, ratio 6), (100, 20, ratio 5), (120, 30, ratio 4)]

Step 3: Start adding items to the knapsack:

1. Add item 1 (60, 10) to the knapsack, remaining capacity = 50 - 10 = 40, total value = 60.
2. Add item 2 (100, 20) to the knapsack, remaining capacity = 40 - 20 = 20, total value = 160.
3. Take a fraction of item 3, 20/30 = 2/3 of item 3. Value added = 120 * (2/3) = 80, total value = 240.

Output: The maximum value that can be obtained is 240.