Week -1

Arrays:

- In Python , Arrays doesn't have any built-in support .Lists are used in place of Arrays.
- A Python list is a dynamic array that can store elements of different data types and allow various operations like appending, slicing, and modifying elements.
- Python also provides the **array module**. Arrays created using this module are more memory-efficient than lists, but they are less flexible.

Lists :

- A Python list is a dynamic array that can store elements of different data types, grow in size, and allow various operations like appending, slicing, and modifying elements.

Key Features of Python Lists:

- Dynamic Size: You can add or remove elements freely.
- Heterogeneous Elements: Python lists can store elements of different data types.
- Indexed Access: Elements are accessed by their index, starting from 0.

Example code:

```
arr= [1,39,20,38,10]
print ("Values in list:", arr)
arr.insert(5,78)
print ("New values in list :",arr)
arr.remove(38)
print ("New values in list after removing :",arr)
del arr[1]
print("After deleting element at index 1:", arr)
print("Traversing the list:")
for i in arr:
    print(i)
```

If importing array module :

```
import array
arr = array.array('i', [10, 20, 30, 40])          #Creating an array of
integers
print("Element at index 1:", arr[1])
arr.append(50)                                              # Inserting 50 to
the array
print("After appending 50:", arr.tolist())                  # Convert to list
for printing, Output: [10, 20, 30, 40, 50]
arr.remove(20)                                              # Removing 20 from
the array
print("After removing 20:", arr.tolist())                   # Convert to list
for printing, Output: [10, 30, 40, 50]
print("Traversing the array:")                              # Traversing the
array
for i in arr:
    print(i)
```

Basic problem on Arrays:

```python
#Find the maximum product of two elements in array
def max_product(arr):
    arr.sort()
    return max(arr[0] * arr[1], arr[-1] * arr[-2])
arr = [1, 20, -5, 4, -6]
print(max_product(arr))
```

```python
#Find the missing number in an array
def find_missing(arr):
#Calculate the expected sum of numbers from 1 to n+1.
#Subtract the actual sum of the array from the expected sum to get the
missing number.
    n = len(arr) + 1
    total_sum = n * (n + 1) // 2
    arr_sum = sum(arr)
    return total_sum - arr_sum
arr = [1, 2, 4, 6, 3, 7, 8]
print(find_missing(arr))
```

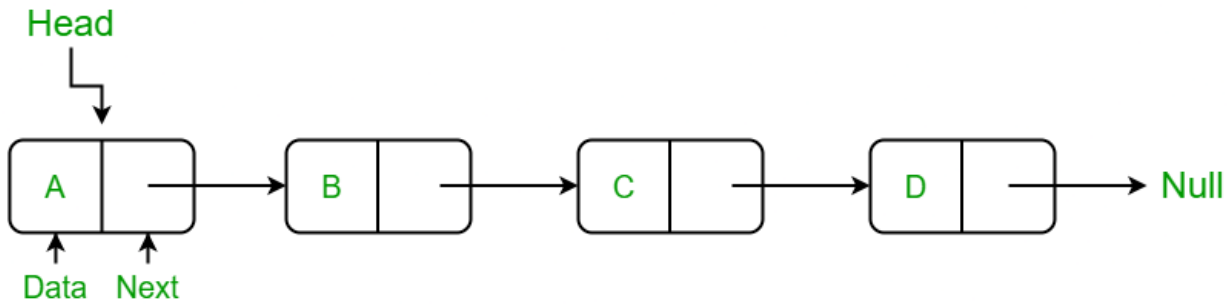
```python
#26. Remove Duplicates from Sorted Array
class Solution:
    def removeDuplicates(self,nums):
        """
        This method removes duplicates from a sorted list `nums` and returns
the number of unique elements.
        """
        if not nums:
            return 0  # If the list is empty, there are no unique elements

        # Initialize the pointer `i` which will keep track of where to place
the next unique element
        i = 0

        # Traverse the list starting from the second element
        for j in range(1, len(nums)):
            # If the current element is different from the last unique
element
            if nums[j] != nums[i]:
                i += 1  # Move `i` to the next position
                nums[i] = nums[j]  # Update the position with the new unique
element

        # Return the count of unique elements
        return i + 1

nums = [1, 1, 2, 2, 3]

    # Create an instance of the Solution class
solution = Solution()

    # Call the removeDuplicates method
k = solution.removeDuplicates(nums)
```

```
    # Print the result
print(f"Number of unique elements: {k}")  # Output: 3
print(f"Modified array: {nums[:k]}")  # Output: [1, 2, 3]
```

```
#Given an array of integers nums and an integer target, return indices of the
two numbers such that they add up to target
class Solution(object):
    def twoSum(self, nums, target):
        num_dict = {}

        for i, num in enumerate(nums):
            complement = target - num

            if complement in num_dict:
                return [num_dict[complement], i]

            num_dict[num] = i


nums1 = [2, 7, 11, 15]
target1 = 9
solution = Solution()
print(solution.twoSum(nums1, target1))
```

Linked List:

- Python does not have a built-in linked list library in its standard library. However, linked lists can be implemented using classes and references in Python.
- A linked list is a linear data structure where elements are stored in nodes. Each node contains data and a reference to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations.



Basic Operations:
1.Creation
2.Insertion
3.Deletion

4.Traversal

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def delete_value(self, value):
        current = self.head
        previous = None
        while current and current.value != value:
            previous = current
            current = current.next
        if current is None:
            print(f"Value {value} not found in the list.")
            return
        if previous is None:
            self.head = current.next
        else:
            previous.next = current.next
        current = None

    def traverse(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.value)
            current = current.next
        return elements


if __name__ == "__main__":
    linked_list = LinkedList()

    # Insert elements
    linked_list.insert_at_beginning(10)
    linked_list.insert_at_beginning(20)
```

```
    linked_list.insert_at_end(30)


    print("Linked List:", linked_list.traverse())

    linked_list.delete_value(10)
    print("Linked List after deletion:", linked_list.traverse())


    linked_list.delete_value(40)
```

Types of Linked Lists:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Example Code:
Singly Linked list :

```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

# Create nodes
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)

# Link nodes
node1.next = node2
node2.next = node3

# Traverse and print the list
current = node1
while current:
    print(current.value, end=" -> ")
    current = current.next
print("None")
```

Doubly Linked List:

```
class DoublyListNode:
    def __init__(self, value=0, next=None, prev=None):
        self.value = value
        self.next = next
        self.prev = prev

# Create nodes
node1 = DoublyListNode(1)
node2 = DoublyListNode(2)
node3 = DoublyListNode(3)
```

```
# Link nodes
node1.next = node2
node2.prev = node1
node2.next = node3
node3.prev = node2

# Traverse forward and print the list
current = node1
while current:
    print(current.value, end=" <-> ")
    current = current.next
print("None")
```

Problems with Arrays :

1. Fixed Size
2. Insertion and Deletion Cost
3. Resizing Overhead
4. Memory Wastage
5. Complexity in Handling Dynamic Data

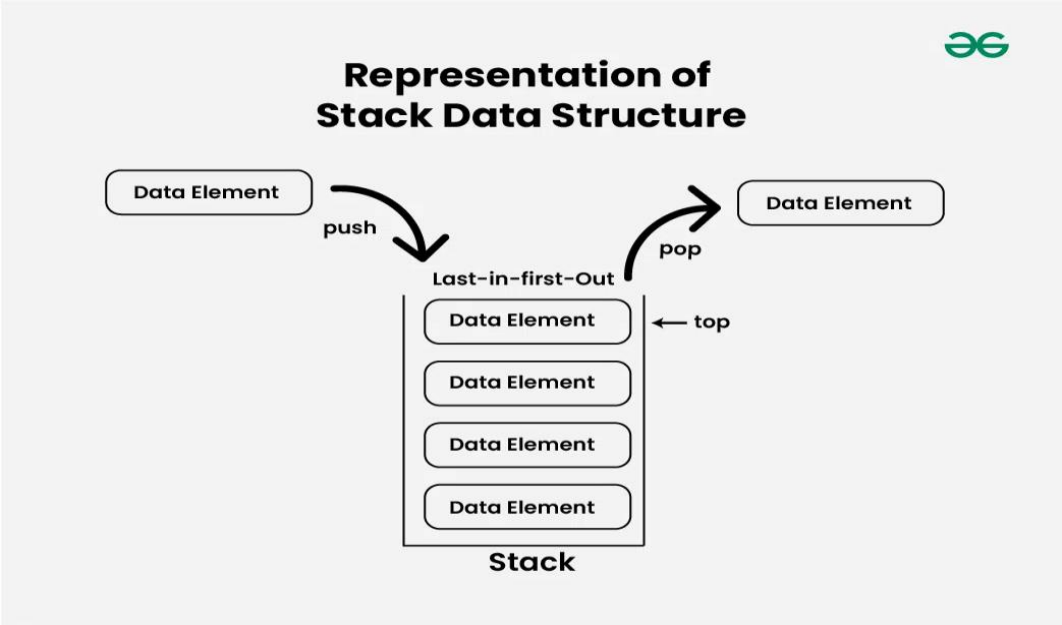
These problems can be resolved when Linked list is used:

1. Linked lists can dynamically grow and shrink as needed.
2. Linked lists allow efficient insertions and deletions. You only need to update pointers or references between nodes.
3. Linked lists do not require resizing. Memory is allocated for each node as needed, so there's no need to move or copy elements.
4. Linked lists allocate memory only for the nodes that are actually used. Although each node has additional overhead for pointers, the overall memory usage is more efficient if the number of elements frequently changes.
5. Nodes can be added or removed without affecting the rest of the list, which simplifies data management and avoids the complexities of resizing.

Stack:

- A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.
- Insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.
- To implement the stack, it is required to maintain the pointer to the top of the stack, which is the last element to be inserted because we can access the elements only on the top of the stack.

There are two types of Stacks:
1.Fixed Stack (fixed size)
2. Dynamic Stack(No fixed size)



Basic Operations :

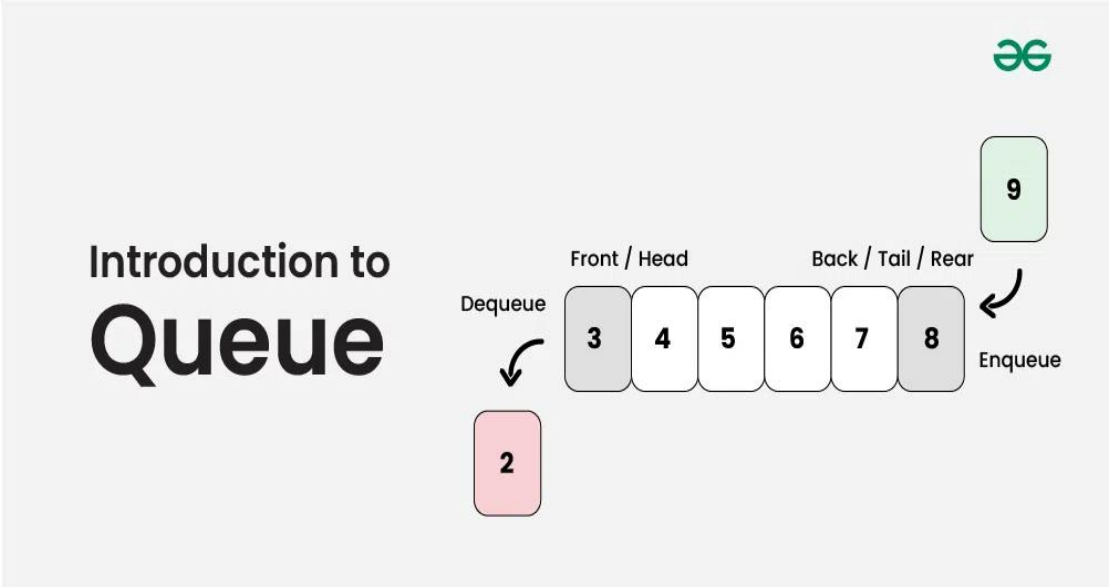
1. Push: Add an element to the top of the stack.
2. Pop: Remove the top element from the stack.
3. Peek/Top: View the top element without removing it.
4. is Empty: Check if the stack is empty.
5. Is Full : Check if Stack is full.

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, element):
        self.stack.append(element)
    def pop(self):
        if self.is_empty():
            return "Stack is empty"
        return self.stack.pop()
    def peek(self):
        if self.is_empty():
            return "Stack is empty"
        return self.stack[-1]
    def is_empty(self):
        return len(self.stack) == 0
    def display(self):
        return self.stack


stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
```

```
print(stack.display())
print(stack.peek())
print(stack.pop())
print(stack.display())
print(stack.pop())
print(stack.display())
print(stack.pop())
print(stack.pop())
```

Queue:

- A queue is a linear data structure that follows the First In, First Out (FIFO) principle.
- First element added to the queue will be the first one to be removed.



Basic Operations:

1. Enqueue: Add an element to the back of the queue.
2. Dequeue: Remove the front element from the queue.
3. Peek/Front: View the front element without removing it.
4. isEmpty: Check if the queue is empty.
5. Rear() : An element is returned which is at rear end without removing it.

```
class Queue:
    def __init__(self):
        self.queue = []


    def enqueue(self, element):
        self.queue.append(element)
        print(f"Enqueued: {element}")
```

```
    def dequeue(self):
        if not self.is_empty():
            dequeued_element = self.queue.pop(0)
            print(f"Dequeued: {dequeued_element}")
            return dequeued_element
        else:
            print("Queue is empty, cannot dequeue.")
            return None


    def peek(self):
        if not self.is_empty():

            print(f"Front element is: {self.queue[0]}")
            return self.queue[0]
        else:
            print("Queue is empty.")
            return None


    def is_empty(self):
        return len(self.queue) == 0


    def display(self):
        print(f"Queue: {self.queue}")


queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
queue.display()
queue.peek()
queue.dequeue()
queue.display()
queue.dequeue()
queue.display()
queue.dequeue()
queue.dequeue()
```

387. First Unique Character in a String :

```
from collections import deque


def first_non_repeating_char_index(s):
    char_count = {}
    queue = deque()


    for index, char in enumerate(s):

        if char in char_count:
```

```
            char_count[char] += 1
        else:
            char_count[char] = 1
            queue.append((char, index))


    while queue:
        char, index = queue.popleft()
        if char_count[char] == 1:
            return index


    return -1



s = "loveleetcode"
print(first_non_repeating_char_index(s))
```

```
#2073. Time Needed to Buy Tickets
class Solution(object):
    def timeRequiredToBuy(self, tickets, k):

        total_time = 0
        for i in range(len(tickets)):

            if tickets[i] > tickets[k]:
                total_time += tickets[k]
            else:
                total_time += tickets[i]


        return total_time



solution = Solution()
tickets = [2, 3, 2]
k = 2
print(solution.timeRequiredToBuy(tickets, k))
```

Hash Tables:

- A Hash table is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on Hashing, where each key is translated by a hash function into a distinct index in an array.
- The process of or decision of storing elements in hash table based on function is called as hashing.
- The function which is used for hashing is called as hash function .
- Genarally , with high load factor leads to collision in hash tables.
- when two or more keys point to the same array index then collision happens. Chaining, open addressing, and double hashing are a few techniques for resolving collisions.

- For lookup, insertion, and deletion operations, hash tables have an average-case time complexity of O(1). Yet, these operations may, in the worst case, require O(n) time, where n is the number of elements in the table.

```
hash_table = {}


hash_table['name'] = 'Alice'
hash_table['age'] = 30
hash_table['city'] = 'New York'


print(hash_table['name'])


del hash_table['age']


print('age' in hash_table)


for key, value in hash_table.items():
    print(f"{key}: {value}")
```

Searching and Sorting Algorithms:

There are different types of searching algorithms:
But most used searching algorithms are:
1.Linear Search: The most basic type of searching algorithm is linear search. It goes through the list element by element until it locates the desired value.

- Start from the first element of the list.It starts searching from first element of the list and compare each element with target value . If it is matched then it returns index else returns -1.

Time Complexity:

- Worst case: O(n)
- Best case: O(1) (if the target is the first element).

```
def linear_search(arr, target):


    for index, value in enumerate(arr):

        if value == target:
            return index
    return -1
if __name__ == "__main__":
    my_list = [10, 20, 30, 40, 50]
    target_value = 30
    result = linear_search(my_list, target_value)
    if result != -1:
```

```
            print(f"Target found at index {result}.")
        else:
            print("Target not found.")
```

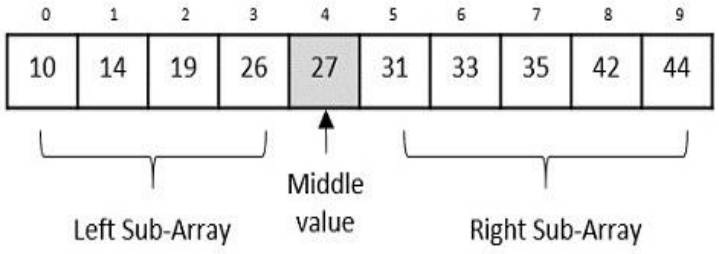
```
#Find second largest
class Solution:
    def print2largest(self, arr):
        largest = second_largest = float('-inf')


        for value in arr:
            if value > largest:

                second_largest = largest
                largest = value
            elif value > second_largest and value < largest:

                second_largest = value


        return second_largest if second_largest != float('-inf') else -1
```

2. Binary Search Algorithm **:**

- Binary Search Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).
- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.

Time Complexity:

- Worst case: O(logn)
- Best case: O(1) (if the middle element is the target).



```
Procedure binary_search:
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while
end procedure
```

```python
def binary_search(a, low, high, key):
   mid = (low + high) // 2
   if (low <= high):
      if(a[mid] == key):
         print("The element is present at index:", mid)
      elif(key < a[mid]):
         binary_search(a, low, mid-1, key)
      elif (a[mid] < key):
         binary_search(a, mid+1, high, key)
   if(low > high):
      print("Unsuccessful Search")

a = [6, 12, 14, 18, 22, 39, 55, 182]
n = len(a)
low = 0
high = n-1
key = 22
binary_search(a, low, high, key)
key = 54
binary_search(a, low, high, key)
```

Sorting Algorithms:

There are different types of Sorting Algorithms:

1.Selection Sort
2.Bubble Sort
3. Insertion Sort
4.Merge Sort
5.Quick Sort
6.Heap Sort

Selection Sort : Selection Sort is a straightforward algorithm that sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and placing it in the sorted portion.

How Selection Sort Works:

1. Start with the first element of the array.
2. Find the smallest element in the unsorted part of the array.
3. Swap this smallest element with the first unsorted element.
4. Move the boundary between the sorted and unsorted parts one element to the right.
5. Repeat steps 2–4 until the entire array is sorted.

Time Complexity:

- Outer loop (to pick each element): O(n)
- Inner loop (to find the minimum element): O(n)
- Overall complexity: $O(n) \times O(n) = O(n^2)$

Advantages:

- Easy to understand and implement.
- Works well for small datasets.

Disadvantages:

- Inefficient for large datasets due to its time complexity of $O(n^2)$
- Not stable, meaning elements with equal values may not retain their relative order.

```
def find_min_index(arr, start):

    min_idx = start
    for i in range(start + 1, len(arr)):
        if arr[i] < arr[min_idx]:
            min_idx = i
    return min_idx
def selection_sort(arr):

    for i in range(len(arr)):
        min_idx = find_min_index(arr, i)

        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr
arr = [29, 10, 14, 37, 13]
sorted_arr = selection_sort(arr)
print("Sorted array:", sorted_arr)
```

Bubble Sort: Bubble Sort is a basic sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It progressively moves the largest element to its correct position by "bubbling" it to the top (rightmost position).

Key Characteristics:

- Simple to understand and implement.
- In-place algorithm (requires no extra memory).
- Stable sorting algorithm (preserves the order of equal elements).

How Bubble Sort Works:

1. Begin at the start of the list.
2. Compare each adjacent pair of elements.
3. If the first element is greater than the second, swap them.
4. Continue comparing and swapping adjacent elements through the list.
5. The largest element moves to the rightmost end of the list.
6. Repeat this process, ignoring the last sorted element, until no swaps are needed.

Process Overview:

- After each pass, the largest unsorted element is placed at the end.
- The next pass focuses on the remaining unsorted portion of the array.
- The process continues until the entire array is sorted.

Time Complexity:

- Worst-case: $O(^{n2)}$ due to nested loops.
- Best-case: O(n) if the array is already sorted and no swaps occur.

Advantages:

- Simple and intuitive.
- In-place sorting (no extra memory required).
- Stable sorting algorithm.

Disadvantages:

- Inefficient for large datasets due to its $O(^{n2})$ time complexity.
- Numerous swaps can lead to high overhead in terms of time, making it costly for large datasets.

```
def bubble_sort(arr):
    n = len(arr)


    for i in range(n):

        swapped = False


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

            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True


        if not swapped:
            break
```

```
    return arr
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

Insertion Sort : Insertion Sort works by gradually building a sorted portion of the list, inserting each unsorted element into its correct position. It's intuitive and stable, maintaining the relative order of equal elements.

How Insertion Sort Works:

1. Begin with the second element (assuming the first is already sorted).
2. Compare this element with the one before it, and insert it into the correct position by swapping.
3. Move to the next element and compare it with all previous sorted elements, swapping until it's in the right place.
4. Repeat this process for each element until the list is fully sorted.

Process Overview:

- For each element, it is compared to the ones before it, and swapped into its correct position in the sorted section.
- This ensures the left portion of the list remains sorted as the right side is processed.

Time Complexity:

- Worst-case: $O(^{n2})$ due to nested comparisons and swaps.
- Best-case: O(n) if the list is already sorted.

Advantages:

- Stable sorting algorithm.
- Efficient for small or nearly sorted lists.
- In-place sorting, no extra memory required.

Disadvantages:

- Inefficient for large lists due to $O(^{n2})$ time complexity.
- Less efficient than other algorithms (like Quick Sort or Merge Sort) for larger datasets.

```
def insertion_sort(arr):

    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1


        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1


        arr[j + 1] = key

    return arr
arr = [12, 11, 13, 5, 6]
sorted_arr = insertion_sort(arr)
print("Sorted array:", sorted_arr)
```

Merge Sort : Merge Sort is a powerful, stable sorting algorithm that uses the divide-and-conquer approach. It recursively splits the input array into smaller subarrays, sorts them, and then merges them back together to form the sorted array.

How Merge Sort Works:

1. Divide: Split the unsorted array into two halves.
2. Recursively divide: Keep dividing the subarrays until each subarray has only one element.
3. Merge: Merge the sorted subarrays by comparing their elements and placing the smaller element first.
4. Repeat: Continue merging until the entire array is sorted.

Process Overview:

- The array is recursively split in half until individual elements are left.
- These elements are merged in sorted order until the entire array is combined back together in sorted form.

Time Complexity:

- Worst-case and average-case: O(nlogn)
- This makes it one of the most efficient sorting algorithms for large datasets.

Advantages:

- Stable sorting algorithm.
- Performs well on large datasets.
- Consistent performance regardless of the input (always O(nlogn)

Disadvantages:

- Requires extra space for the temporary subarrays, making it less space-efficient than in-place algorithms like Quick Sort.

```
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

    return arr
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print("Sorted array:", sorted_arr)
```

Quick Sort: Quick Sort is a highly efficient sorting algorithm that follows the divide-and-conquer approach. It works by partitioning an array into two subarrays around a pivot element, ensuring that elements smaller than the pivot are on the left and elements larger are on the right.

How Quick Sort Works:

1. Select a pivot element from the array.
2. Partition the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
3. Recursively apply the same process to the left and right subarrays until each subarray has one element.
4. Once all partitions are sorted, the elements are combined to form a fully sorted array.

Time Complexity:

- Best and Average case:O(nlogn) when the pivot divides the array into two nearly equal parts.
- Worst-case: $O(^{n2})$ occurs when the pivot is poorly chosen (e.g., always selecting the largest or smallest element in an already sorted array).

Advantages:

- Very efficient for large datasets.
- In-place sorting algorithm, requiring minimal additional memory.
- Low overhead and fast performance in most practical scenarios.

Disadvantages:

- The worst-case performance of $O(^{n2})$ occurs with poor pivot selection.
- Not stable, meaning that equal elements may not maintain their relative order.
- Less efficient on small datasets compared to other algorithms like Insertion Sort.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:

        pivot = arr[len(arr) // 2]


        less = [x for x in arr if x < pivot]
```

```
        equal = [x for x in arr if x == pivot]
        greater = [x for x in arr if x > pivot]


        return quick_sort(less) + equal + quick_sort(greater)
arr = [10, 7, 8, 9, 1, 5]
sorted_arr = quick_sort(arr)
print("Sorted array:", sorted_arr)
```

Heap Sort: Heap Sort is a comparison-based sorting technique that uses a binary heap data structure. It sorts an array by first transforming it into a heap and then repeatedly extracting the maximum element to build the sorted array. It operates similarly to Selection Sort but with more efficient heap operations.

How Heap Sort Works:

1. Build a Heap: Convert the input array into a Max-Heap, where the largest element is at the root.
2. Extract Max: Swap the root (maximum) with the last element in the heap. Remove the last element from the heap (now sorted).
3. Heapify: Restore the heap property for the remaining elements.
4. Repeat: Continue extracting the maximum and heapifying until only one element remains.

Process Overview:

- Heapify: Transforming the array into a heap structure.
- Sort: Repeatedly extracting the maximum element and re-heapifying.

Time Complexity:

- Best, Average, and Worst Case: O(nlog⁡n)O(n \log n)O(nlogn) due to the repeated heap operations.

Advantages:

- Efficient: Time complexity is O(nlog⁡n)O(n \log n)O(nlogn), making it suitable for large datasets.
- In-place Sorting: Requires minimal additional memory.

Disadvantages:

- Higher Constants: Although it has the same asymptotic time complexity as Merge Sort, it can be less efficient in practice due to higher overhead.
- Unstable: Does not preserve the relative order of equal elements.
- Less Efficient for Complex Data: May not be optimal for sorting complex data types.

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        largest = left
    if right < n and arr[largest] < arr[right]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap
        heapify(arr, n, largest)
```

```
def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]  # swap
        heapify(arr, i, 0)

    return arr
arr = [12, 11, 13, 5, 6, 7]
sorted_arr = heap_sort(arr)
print("Sorted array:", sorted_arr)
```

```
#Floor in a Sorted Array

class Solution:
    # User function Template for python3

    # Complete this function
    def findFloor(self, A, N, X):
        # Your code here
        low, high = 0, N - 1
        result = -1

        while low <= high:
            mid = (low + high) // 2

            if A[mid] <= X:
                result = mid  # Potential floor found
                low = mid + 1  # Search in the right half for a larger valid
floor
            else:
                high = mid - 1  # Search in the left half for a smaller valid
floor

        return result
```

Algorithms:

Dijkstra's algorithm

- Dijkstra Algorithm is used to find the shortest path from a starting vertex to all other vertices in a graph with non-negative edge weights.
- It works by iteratively selecting the nearest unvisited vertex and updating the distances to its neighboring vertices.
- This algorithm is widely used in various applications.

Algorithm :

- Set the distance to the source vertex as 0 and all other vertices as infinity.
- Mark all vertices as unvisited.
- Select the unvisited vertex with the smallest distance value and designate it as the current vertex.

- For each unvisited neighbor of the current vertex, calculate the potential new shortest distance from the source. If this new distance is smaller than the known distance, update it.
- After processing the current vertex, mark it as visited. Visited vertices are not revisited.
- Repeat the selection and update process until all vertices have been visited.
- After completing the algorithm, the shortest distance from the source vertex to each vertex in the graph is determined.

Time Complexity:O(V2) where V is the number of vertices in the graph.

Disadvantages:
- Negative Weights: Dijkstra's algorithm does not handle graphs with negative edge weights. For such cases, the Bellman-Ford algorithm is more appropriate.
- Dense Graphs: The algorithm can be inefficient on graphs with a high density of edges.
- High Memory Usage: The algorithm requires storing distance values and maintaining a priority queue, which can be memory-intensive.
- Unweighted Graphs: On unweighted graphs, algorithms like Breadth-First Search (BFS) are more efficient.

Graph Traversal:

Graph traversal refers to the process of visiting vertices in a graph. The goal can be to visit all vertices, or as many as possible, starting from a specific vertex. The two most common traversal techniques are:

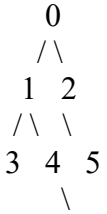
1. Depth First Search (DFS)
2. Breadth First Search (BFS)

Depth First Search (DFS)**:** Depth First Search (DFS) is a graph traversal algorithm that explores a graph by moving as deep as possible along each branch before backtracking.

- DFS can be implemented using either a **stack** or **recursion**.

Algorithm:

- Mark the starting vertex as visited and push it to a stack (in iterative DFS) or call the function recursively (in recursive DFS).
- Pop a vertex from the stack (or return from recursion) and check its unvisited neighbors.
- Visit an unvisited neighbor of the current vertex, mark it as visited, and push it to the stack (or recurse with that vertex).
- Repeat this process until you've visited all vertices.
- Backtrack when no unvisited neighbors remain. This ensures that all vertices are eventually visited, even those that are only reachable via backtracking.



6

1. Start at 0. Mark 0 as visited and push it onto the stack. Stack: [0]
2. Pop 0 from the stack. Visit its adjacent unvisited vertex 1 (arbitrarily chosen). Stack: [1]
3. Pop 1. Visit its unvisited adjacent vertex 3. Stack: [3]
4. Pop 3. No unvisited neighbors, so backtrack to 1. Stack: [1]
5. At 1, visit its next unvisited neighbor 4. Stack: [4]
6. Pop 4. Visit its unvisited adjacent vertex 6. Stack: [6]
7. Pop 6. No unvisited neighbors, backtrack to 4, then 1. Stack: [1]
8. Backtrack to 0 and visit 2. Stack: [2]
9. Pop 2. Visit its unvisited neighbor 5. Stack: [5]
10. Pop 5. No unvisited neighbors, backtrack to 2 and then 0.

Final order of traversal: 0 → 1 → 3 → 4 → 6 → 2 → 5.

```
class GraphDFS:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

    def dfs(self, start):
        visited = set()
        stack = [start]

        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                print(vertex, end=" ")
                visited.add(vertex)


                for neighbor in reversed(self.graph.get(vertex, [])):
                    if neighbor not in visited:
                        stack.append(neighbor)



graph_dfs = GraphDFS()
graph_dfs.add_edge(0, 1)
graph_dfs.add_edge(0, 2)
graph_dfs.add_edge(1, 3)
graph_dfs.add_edge(1, 4)
graph_dfs.add_edge(2, 5)
graph_dfs.add_edge(4, 6)

print("DFS Traversal:")
graph_dfs.dfs(0)
```

Time Complexity: O(V+E)O(V + E)O(V+E), where VVV is the number of vertices and EEE is the number of edges.

Breadth First Search : Breadth-First Search (BFS) is a graph traversal algorithm that explores vertices level by level, starting from a given vertex.

- BFS is particularly useful for finding the shortest path in unweighted graphs, as it processes vertices in increasing order of their distance from the start.

Algorithm:

- Add the Start Vertex to the Queue:
  Begin by inserting the initial vertex (source vertex) into the queue, as this will be the first vertex to be processed.
- Remove and Process the Vertex:
  Take out the vertex at the front of the queue, mark it as visited, and inspect its neighboring vertices.
- Add Unvisited Neighboring Vertices:
  For each unvisited neighboring vertex, mark it as visited and insert it into the back of the queue. This ensures that all vertices at the current level are processed before moving to the next level.
- Continue Until the Queue is Empty:
  Repeat the process of dequeuing and processing vertices until the queue is empty, indicating that all reachable vertices have been visited.

```
from collections import deque

class GraphBFS:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):

        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

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

        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                print(vertex, end=" ")
                visited.add(vertex)


                for neighbor in self.graph.get(vertex, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
graph_bfs = GraphBFS()
graph_bfs.add_edge(0, 1)
graph_bfs.add_edge(0, 2)
graph_bfs.add_edge(1, 3)
graph_bfs.add_edge(1, 4)
graph_bfs.add_edge(2, 5)
```

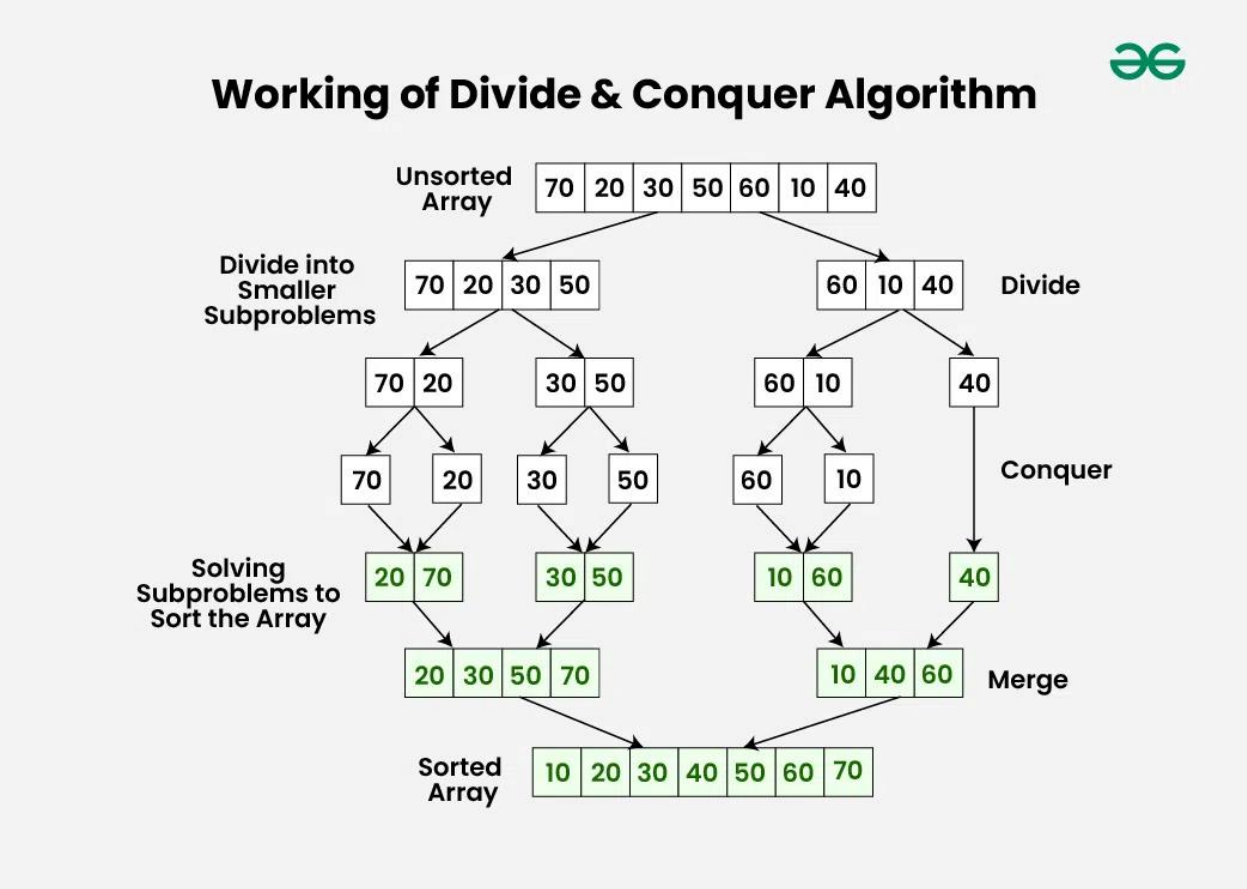
```
graph_bfs.add_edge(3, 6)

print("BFS Traversal:")
graph_bfs.bfs(0)
```

Time Complexity: O(V+E) where:

- V is the number of vertices.
- E is the number of edges.

Divide and Conquer:

- This Algorithm involves breaking a larger problem into smaller subproblems, solving them independently.



Algorithm:

- Divide: The problem is split into smaller subproblems. For example, in sorting, the array is divided into two halves.
- Conquer: The subproblems are solved recursively. In the base case, if the subproblem is small enough (e.g., a single element in sorting), the solution is simple.
- Combine: After solving the subproblems, the solutions are combined to produce the solution to the original problem. In sorting, this could involve merging two sorted arrays

Basic code:

```python
class DivideAndConquer:
    def divide(self, problem):

        subproblem1 = problem[:Len(problem) // 2]
        subproblem2 = problem[len(problem) // 2:]
        return subproblem1, subproblem2

    def conquer(self, subproblem):

        if len(subproblem) == 1:
            return subproblem


        left, right = self.divide(subproblem)
        solved_left = self.conquer(left)
        solved_right = self.conquer(right)


        return self.combine(solved_left, solved_right)

    def combine(self, left, right):

        result = []
        i = j = 0


        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

        result.extend(left[i:])
        result.extend(right[j:])

        return result

    def solve(self, problem):

        return self.conquer(problem)

arr = [38, 27, 43, 3, 9, 82, 10]
dac = DivideAndConquer()
sorted_arr = dac.solve(arr)
print("Sorted array:", sorted_arr)
```

Advantages:

1. Efficient for large problems.
2. Improved time complexity.
3. Supports parallelism.
4. Ideal for recursive problems.
5. Modular and cleaner code.

Disadvantages:

1. Recursion overhead.
2. Higher memory usage.
3. Expensive combine step.
4. Risk of poor pivot selection (in Quick Sort).
5. Less efficient for small problems.

6.
```python
# Find the Maximum element in the list
class MaxElement:
    def find_max(self, arr, low, high):

        if low == high:
            return arr[low]


        mid = (low + high) // 2


        max_left = self.find_max(arr, low, mid)
        max_right = self.find_max(arr, mid + 1, high)

        return max(max_left, max_right)



arr = [1, 45, 23, 89, 76, 12, 4]
finder = MaxElement()
max_value = finder.find_max(arr, 0, len(arr) - 1)
print("Maximum element:", max_value)
```

Divide and Conquer Applications:

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix multiplication
- Karatsuba Algorithm

Fractional Knapsack Problem :

- The Fractional Knapsack Problem involves selecting items to maximize profit, where items can be broken into smaller fractions.
- This problem is optimally solved using a greedy algorithm, which ensures that the solution will always be the maximum possible value that can be carried in the knapsack.

Algorithm:

- List Items: Consider all items with their respective weights and profits.

- Calculate Ratios: Compute the profit-to-weight ratio (Pi/Wi)for each item.
- Sort: Sort items in descending order based on their profit-to-weight ratios.
- Select Items: Add items to the knapsack without exceeding its capacity. If there's remaining capacity but no full items fit, add a fractional part of the next item.
- Maximize Value: This method ensures that the knapsack is filled in a way that maximizes total profit.

Advantages:

- Optimal Solution: The greedy algorithm guarantees the maximum possible value for the knapsack.
- Flexibility: The ability to include fractions of items allows for easier maximization of total value.
- Practical: Mirrors real-world scenarios where resources or investments can be divided.

Disadvantages:

- Indivisible Items: Not suitable for problems where items cannot be divided.
- Realism: In some contexts, the ability to take fractions may oversimplify the problem and lead to impractical solutions.
- Generalization: The greedy approach used here may not apply to other problems that require different strategies.