

# Data Structures and Algorithms Practice

## People Tech Group Week 1(L1 Batch 2)

Name: Shrestha Sai Laxmi Yadavilli  
Email: shrestha.yadavilli22@gmail.com

### Table of Contents

<b>1. Introduction .....</b>	<b>3</b>
1.1 Topics Covered.....	3
1.2 Objectives of the Week's Tasks .....	3
<b>2. Arrays .....</b>	<b>4</b>
2.1 What is an Array?.....	4
2.2 Contiguous Memory Allocation of Array .....	4
2.3 Applications, Benefits and Limitations of an Array .....	5
2.4 Basic Operations in an Array .....	5
2.5 Array Problems - Challenges, Solution and Approach .....	8
2.6 Key Takeaways from the Resources .....	18
<b>3. Linked Lists.....</b>	<b>19</b>
3.1 Introduction to Linked Lists .....	19
3.2 Memory Allocation .....	19
3.3 Applications, Benefits and Limitations of Linked Lists.....	19
3.4 Linked List Vs Arrays .....	20
3.5 Basic Operations.....	20
3.6 Hands-on Coding Exercises .....	23
3.7 Key Takeaways from the Resources .....	26
<b>4. Stacks .....</b>	<b>27</b>
4.1 Introduction to Stacks .....	27
4.2 Memory Allocation .....	27
4.3 Applications, Benefits and Limitations of Stacks .....	27
4.4 Basic Operations.....	27
4.5 Hands-on Coding Exercises .....	28
4.6 Key Takeaways from the Resources .....	32
<b>5. Queues.....</b>	<b>33</b>
5.1 Introduction to Queues .....	33
5.2 Memory Allocation .....	33
5.3 Applications, Benefits and Limitations of Stacks .....	33
5.4 Basic Operations.....	33
5.5 Hands-on Coding Exercises .....	35
5.6 Key Takeaways from the Resources .....	39
<b>6. Hash Tables .....</b>	<b>40</b>
6.1 Introduction to Hash Tables .....	40
6.2 Memory Allocation .....	40
6.3 Applications, Benefits and Limitations of Stacks .....	40
6.4 Basic Operations.....	40
6.5 Hands-on Coding Exercises .....	42
6.6 Key Takeaways from the Resources .....	45
<b>7. Algorithms.....</b>	<b>46</b>
7.1 Searching Algorithms .....	46
7.2 Sorting Algorithms .....	50
7.3 Dijkstra Algorithm and Bellman-Ford Algorithm.....	57

7.4 Graph Traversals(BFS and DFS).....	60
7.5 Divide and Conquer.....	62
7.6 Fractional Knapsack Problem.....	67
<b>8. Reference .....</b>	<b>78</b>

# 1. Introduction

In the first week of the L1 Program, the focus is on mastering fundamental **data structures** and **algorithms**, which are critical for efficient software development. Understanding these concepts is essential because they form the backbone of problem-solving in computer science, allowing developers to manage and manipulate data effectively while optimizing performance. The topics covered during Week 1 include **arrays**, **linked lists**, **stacks**, **queues**, **hash tables**, and **sorting and searching algorithms**, along with **graph algorithms** such as **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**, **Dijkstra's Algorithm**, **Bellman-Ford Algorithm**, **Knapsack Problem** and **Divide and Conquer**. These data structures and algorithms are widely used in various applications, from memory management to processing large datasets efficiently.

## 1.1 Topics Covered

- **Arrays:** A fundamental data structure that stores elements in a contiguous block of memory, allowing for efficient indexing and retrieval of elements.
- **Linked Lists:** A linked list is a linear data structure where elements, called nodes, are connected by pointers, making dynamic memory allocation and efficient insertions/removals possible.
- **Stacks:** A stack is a last-in, first-out (LIFO) data structure where elements are added and removed from the top, often used for function call management, undo operations, and parsing tasks.
- **Queues:** A queue is a first-in, first-out (FIFO) data structure where elements are added to the rear and removed from the front, useful for scheduling processes and managing resources.
- **Hash Tables:** Hash tables store key-value pairs and use a hash function to map keys to indexes in an array, enabling fast lookups, insertions, and deletions.
- **Sorting and Searching Algorithms:** Algorithms designed to arrange data in a particular order (e.g., quicksort, mergesort) or find specific data efficiently (e.g., binary search).
- **Dijkstra Algorithm and Bellman-Ford Algorithm:** Dijkstra's algorithm is a greedy algorithm used to find the shortest path between nodes in a graph with non-negative edge weights. The Bellman-Ford algorithm, in contrast, can handle graphs with negative edge weights and is used to detect negative weight cycles.
- **Graph Algorithms (BFS and DFS):** BFS and DFS are techniques for traversing or searching through graph structures, useful in networking, social networks, and pathfinding problems.
- **Divide and Conquer:** Divide and Conquer is a problem-solving technique that breaks a problem into smaller subproblems, solves them independently, and then combines the solutions to solve the original problem. Commonly used in algorithms like Merge Sort and Quick Sort.
- **Fractional Knapsack Problem:** The Fractional Knapsack Problem is an optimization problem where you can take fractions of items, aiming to maximize the total value in a knapsack without exceeding its weight limit. It is typically solved using a greedy algorithm.

## 1.2 Objectives of the Week's Tasks

- Gain a solid understanding of the **basic operations** of data structures such as creation, insertion, deletion, and traversal.
- Implement and manipulate **arrays** and **linked lists** to strengthen foundational knowledge.
- Understand the **performance trade-offs** between arrays and linked lists in terms of memory allocation, access time, and practical use cases.
- Develop a hands-on understanding of **stacks**, **queues**, and **hash tables** by implementing their operations and solving Leetcode coding problems where they are applicable.
- Learn **sorting and searching algorithms** to better understand how data is arranged and retrieved efficiently in large datasets.
- Learning about **graph traversal algorithms** like BFS and DFS to solve problems that involve searching through complex data structures like networks or trees.
- Solve practice problems on platforms like LeetCode or HackerRank to apply the learned concepts in real coding challenges.
- Document the **learnings**, **challenges**, and **solutions** encountered during the week to reflect on the progress and areas for improvement.

## 2. Arrays

### 2.1 What is an Array?

**Key Concepts:** An array is a data structure that stores elements in contiguous memory locations, meaning each element can be efficiently accessed using an index. In an array, all elements are of the same data type. They are widely used in computer science due to their simplicity and efficiency in storing data sequentially.

- **Structure:** Arrays store elements in adjacent memory blocks, where each element is identified by a unique index. The first element is at index 0, and the last element is at index  $n-1$  for an array of size  $n$ .

An array could be considered a row of boxes where every box can store some element of the same data type.

- **Array Index:** Elements are identified in the array by their indexes. (It starts from 0)
- **Array Element:** The items stored in the array are called element.
- **Array Length:** The size of the array or the number of elements present in the array is the length of the array.

Below is an example of an array of size 7. The range of indexes is always  $[0, \text{size}-1]$ . An array element can be directly accessed using the index value.

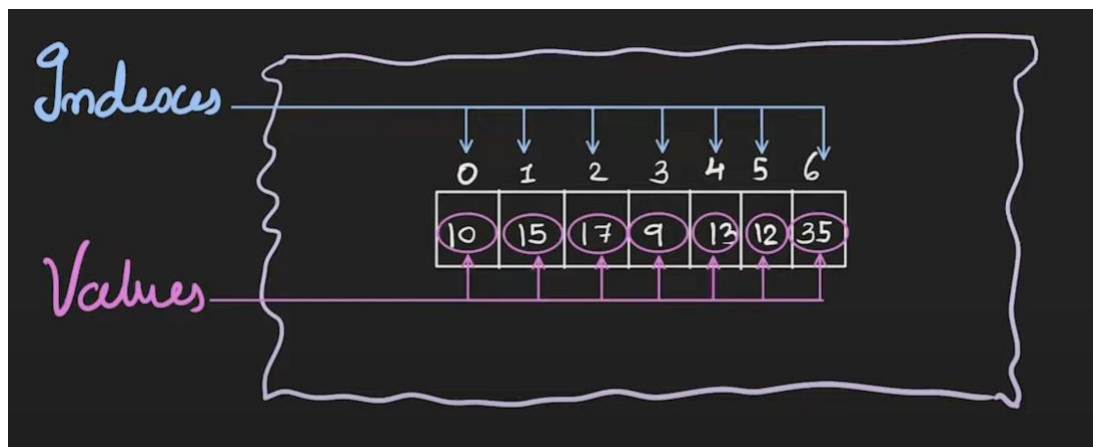


Figure 1: Example of an Array of size 7

### 2.2 Contiguous Memory Allocation of Array

A Single contiguous section/part of memory is allocated to the array. In the example given in Figure 2, there is an integer array of size  $n$ . When an element '10' is added to the index 0, it occupies the memory between 1000 and 1003 as the int size is 4 bytes. The next element will be allocated to the next continuous memory which will be 1004-1007 (4 bytes). The element in index 2 will have the memory between 1008-1011 and so on. This is the contiguous memory allocation of an array which means elements will be allocated sequentially in memory. The location of the next index depends on the data type we use.

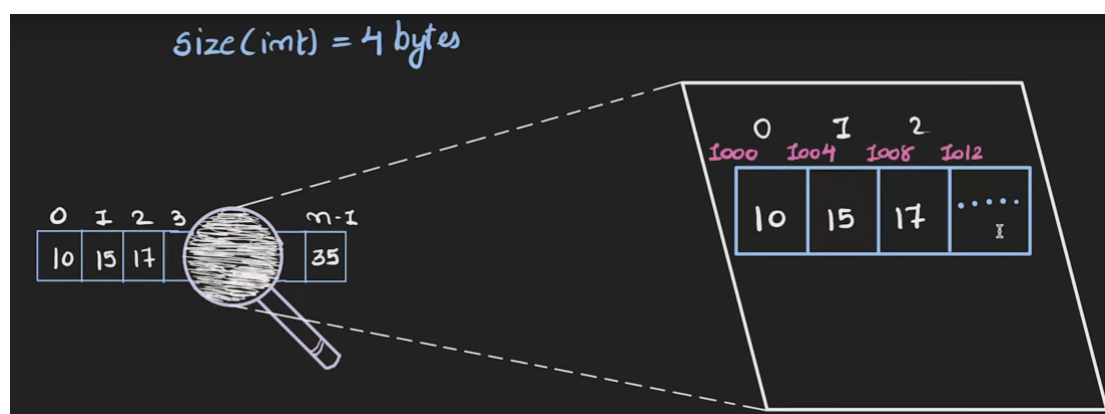


Figure 2: Contiguous Memory Allocation of Array

Arrays are divided into static and dynamic arrays based on memory allocation.

- Static array allocates the memory at compile time and its size is fixed.
- Dynamic array automatically grows when we try to make insertion and there is no space left for new item.

In C language, an array is a static array so once the size is allocated it cannot be changed but in Python arrays are dynamic which are called lists.

Arrays can also be divided into two categories based on dimension. A one-dimensional array is an array in which elements are stored in a single row of elements and a multi-dimensional array has elements which are stored in multiple rows

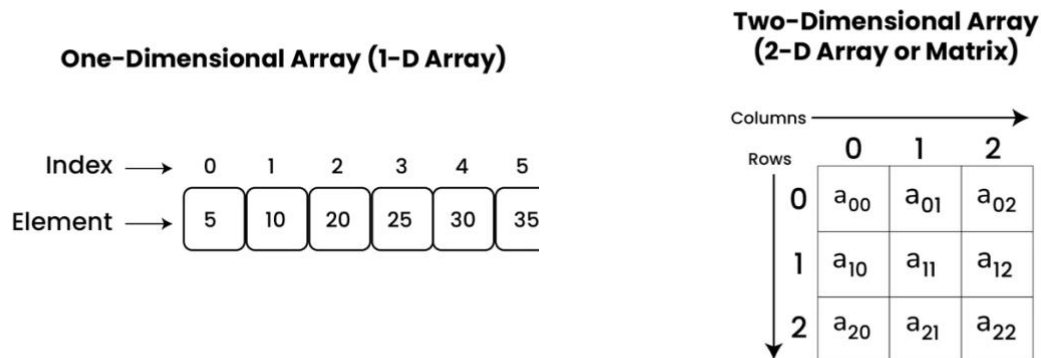


Figure 3: Source - [geeksforgeeks.com](https://www.geeksforgeeks.com/)

Figure 4: Source - [geeksforgeeks.com](https://www.geeksforgeeks.com/)

## 2.3 Applications, Benefits and Limitations of an Array

### Applications:

- Arrays can be used to store and retrieve data in a specific order for processing
- Arrays can be used as a foundation data structure to implement stacks and queues
- Arrays can be used to sort the data and search for a specific element
- Arrays could be used for creating dynamic data structures such as linked lists or trees.
- Dynamic Programming algorithms use arrays to store intermediate results.
- Using arrays, data can be represented as tables and matrices to perform mathematical computations like matrix multiplication, linear algebra etc.

### Benefits:

- Arrays allow for **O(1)** time complexity when accessing elements by their index, making them ideal for applications where random access to data is necessary.
- Since all elements are of the same type and stored contiguously, arrays offer efficient memory utilization compared to other data structures.
- Arrays are simple to understand and implement. They are often the go-to data structure for storing and retrieving a fixed amount of data.

### Limitations:

- In some programming languages like C, once an array is declared, its size cannot be changed. This is a significant limitation when dealing with dynamic datasets where the number of elements can vary.
- Inserting or deleting an element from the middle of an array involves shifting elements, resulting in an **O(n)** time complexity, where  $n$  is the number of elements in the array.
- Arrays need contiguous blocks of memory, which can be problematic when dealing with large datasets or in systems with fragmented memory.

## 2.4 Basic Operations in an Array

Common operations performed on arrays include the following:

## 1. Creation:

- In Python, arrays are created in the form of lists. These lists could be empty or have some values in it.

```
list1 = [] #Initializing the variable list1 as an empty list
list2 = [1, 2, 3, 4, 5] #Initializing list list2 with integer values(length = 5)
list3 = ['cat', 'dog', 'mouse'] #Initializing list list3 with string values(length = 3)
```

## 2. Insertion:

- Insertion can happen at the beginning, middle, or end of an array. Inserting at the end is efficient ( $O(1)$ ), but inserting elsewhere requires shifting elements, resulting in  $O(n)$  complexity.

Inserting one element at the end using `append()`

```
arr = [1, 2, 3]
arr.append(4) # Insert 4 at the end
print(arr) # Output: [1, 2, 3, 4]
```

Inserting multiple elements at the end using `extend()`

```
arr = [1, 2, 3]
arr.extend([4, 5, 6]) # Insert multiple elements [4, 5, 6] at the end
print(arr) # Output: [1, 2, 3, 4, 5, 6]
```

Inserting one element at the beginning using `insert()`

```
arr = [1, 2, 3]
arr.insert(0, 0) # Insert 0 at the beginning
print(arr) # Output: [0, 1, 2, 3]
```

Inserting multiple elements at the beginning using slicing

```
arr = [1, 2, 3]
arr = [0, -1] + arr # Insert multiple elements [0, -1] at the beginning
print(arr) # Output: [0, -1, 1, 2, 3]
```

Inserting multiple elements at the beginning using `insert` in a loop

```
arr = [1, 2, 3]
for i, elem in enumerate([-1, 0]):
    arr.insert(i, elem)
print(arr) # Output: [-1, 0, 1, 2, 3]
```

Inserting one element at a specific index

```
arr = [1, 2, 3]
arr.insert(1, 5) # Insert 5 at index 1
print(arr) # Output: [1, 5, 2, 3]
```

Inserting multiple elements at a specific index

```
arr = [1, 2, 3]
arr[1:1] = [5, 6] # Insert multiple elements [5, 6] at index 1
print(arr) # Output: [1, 5, 6, 2, 3]
```

### 3. Deletion:

- Similar to insertion, deleting an element requires shifting the remaining elements. This operation takes  $O(n)$  time unless the element is at the end of the array.

Deleting one element from a specific index

```
arr = [1, 2, 3, 4, 5]
del arr[2] # Delete element at index 2 (the number 3)
print(arr) # Output: [1, 2, 4, 5]
```

Deleting first occurrence of a value using remove

```
arr = [1, 2, 3, 4, 5]
arr.remove(3) # Remove the first occurrence of the value 3
print(arr) # Output: [1, 2, 4, 5]
```

Deleting multiple elements by value

```
arr = [1, 2, 3, 3, 4, 5, 3]
arr = [x for x in arr if x != 3] # Remove all occurrences of the value 3
print(arr) # Output: [1, 2, 4, 5]
```

Deleting an element and displaying the value(pop)

By Index Value:

```
arr = [1, 2, 3, 4, 5]
popped_value = arr.pop(2) # Remove element at index 2 and return it
print(popped_value) # Output: 3
print(arr) # Output: [1, 2, 4, 5]
```

Last element:

```
arr = [1, 2, 3, 4, 5]
popped_value = arr.pop() # Remove and return the last element
print(popped_value) # Output: 5
print(arr) # Output: [1, 2, 3, 4]
```

Deleting multiple elements by index

```
arr = [1, 2, 3, 4, 5, 6]
del arr[1:4] # Remove elements from index 1 to 3 (2, 3, 4)
print(arr) # Output: [1, 5, 6]
```

Deleting all elements using clear()

```
arr = [1, 2, 3, 4, 5]
arr.clear() # Remove all elements
print(arr) # Output: []
```

Deleting elements conditionally

```
arr = [1, 2, 3, 4, 5, 6, 7, 8]
arr = [x for x in arr if x % 2 == 0] # Remove all odd numbers
print(arr) # Output: [2, 4, 6, 8]
```

Deleting elements from the beginning or end using slicing

```
arr = [1, 2, 3, 4, 5]
arr = arr[2:] # Remove the first two elements
print(arr) # Output: [3, 4, 5]

arr = [1, 2, 3, 4, 5]
arr = arr[:-2] # Remove the last two elements
print(arr) # Output: [1, 2, 3]
```

#### 4. Traversal:

- Traversing an array means visiting each element one by one, typically using a loop. This operation has a time complexity of  $O(n)$ , where  $n$  is the number of elements in the array.

for loop

```
arr = [1, 2, 3, 4, 5]
for i in range(len(arr)): # Traverse using index
    print(arr[i])
```

```
arr = [1, 2, 3, 4, 5]
for element in arr: # Traverse by directly accessing elements
    print(element)
```

while loop

```
arr = [1, 2, 3, 4, 5]
i = 0
while i < len(arr):
    print(arr[i])
    i += 1
```

## 2.5 Array Problems - Challenges, Solution and Approach

### 2.5.1 Problem 1: Reverse a subarray from a given array

#### Problem Statement([Link](#)):

Given an array **arr**, you need to reverse a subarray of that array. The range of this subarray is given by indices **L** and **R** (1-based indexing).

#### Challenges Faced:

- Python uses 0-based indexing for arrays and lists, while the problem specifies 1-based indexing. This requires an adjustment when accessing the array elements to correctly refer to the desired subarray.
- It's important to ensure that the given indices **L** and **R** are within the valid bounds of the array to avoid index out-of-range errors.
- To reverse the elements of the subarray in-place without allocating extra space, slicing and Python's built-in list reversal need to be utilized.

#### Approach:

- First, convert the given indices **L** and **R** from 1-based indexing to 0-based indexing by subtracting 1 from both values. This allows direct manipulation of the Python array.
- In Python, extracting and manipulating subarrays is easy using slice notation (`arr[start:end]`). For example, `arr[L:R+1]` retrieves the subarray between indices **L** and **R** (both inclusive).



- The subarray can be reversed in place by assigning `arr[L:R+1] = arr[L:R+1][::-1]`. This creates a reversed copy of the subarray and places it back into the original array.
- Edge Cases: If `L == R`, no reversal is needed, as the subarray consists of just one element. If `L` and `R` are out of bounds or if `L > R`, array is returned unchanged.

### Solution:

**Reverse sub array**

Difficulty: Basic Accuracy: 31.78% Submissions: 26K+ Points: 1

Given an array `arr`, you need to reverse a subarray of that array. The range of this subarray is given by indices `L` and `R` (1-based indexing).

**Examples:**

**Input:** `arr` = [1, 2, 3, 4, 5, 6, 7] and `L` = 2 and `R` = 4

**Output:** [1, 4, 3, 2, 5, 6, 7]

**Explanation:** After reversing the elements in range 2 to 4 (2, 3, 4), modified array is 1, 4, 3, 2, 5, 6, 7.

**Input:** `arr` = [1, 6, 7, 4] and `L` = 1 and `R` = 4

**Output:** [4, 7, 6, 1]

**Explanation:** After reversing the elements in range 1 to 4 (1, 6, 7, 4), modified array is 4, 7, 6, 1

```

1 #User function Template for python3
2 class Solution:
3     def reverseSubArray(self, arr, l, r):
4         # code here
5         l -= 1
6         r -= 1
7         arr[l:r+1] = arr[l:r+1][::-1]
8         return arr
9
10
11

```

### Code:

```

def reverseSubArray(arr,l,r):

    l -=1

    r -=1

    arr[l:r+1] = arr[l:r+1][::-1]

    return arr

```

### Example:

```

arr = [1, 2, 3, 4, 5, 6, 7]

l = 2

r = 4

reverseSubArray(arr,l,r) # Output: [1, 4, 3, 2, 5, 6, 7]

```

## 2.5.2 Problem 2: Array Leaders

### Problem Statement([Link](#)):

Given an array `arr` of `n` positive integers, your task is to find all the leaders in the array. An element of the array is considered a leader if it is greater than all the elements on its right side or if it is equal to the maximum element on its right side. The rightmost element is always a leader.

### Challenges Faced:

- Since an element is compared with all elements to its right, a brute-force approach would involve multiple comparisons, which could lead to an inefficient solution with a nested loop resulting in a time complexity of  $O(n^2)$ . Optimizing this comparison is key to reducing the overall runtime.
- At each index, it has to be efficiently determined whether the current element is greater than or equal to all the elements on its right.
- Leaders need to be stored in the order they appear in the array, but since we are iterating from the right to the left, the leaders will be collected in reverse order. Handling this by reversing the result list at the end ensures the correct order.

### Approach:

- First a variable `max` is initialized to keep track of the maximum element and an empty list called `result_list` is initialized.
- The array is iterated through in a reverse order starting from the rightmost element. The rightmost element is always a leader, hence, first the array is iterated through from right to left.

- The code keeps track of the maximum element (max) encountered so far, starting from the rightmost element. While iterating if the current element is greater than or equal to the maximum found then that element is a leader.
- These elements which are found as the leaders are appended to the result\_list.
- Since the array is iterated from right to left, the leaders are naturally appended in reverse order. After the loop, the result\_list is reversed to restore the correct order.

### Time Complexity:

The approach only requires a single pass through the array, achieving  $O(n)$  time complexity, which is optimal for this problem.

### Solution:

**Array Leaders**

Difficulty: Easy Accuracy: 29.94% Submissions: 723K+ Points: 2

Given an array `arr` of `n` positive integers, your task is to find all the leaders in the array. An element of the array is considered a leader if it is greater than all the elements on its right side or if it is equal to the maximum element on its right side. The rightmost element is always a leader.

**Examples**

Input: `n = 6, arr[] = {16,17,4,3,5,2}`  
 Output: 17 5 2  
 Explanation: Note that there is nothing greater on the right side of 17, 5 and, 2.

Input: `n = 5, arr[] = {10,4,2,4,1}`  
 Output: 10 4 4 1  
 Explanation: Note that both of the 4s are in output, as to be a leader an equal element is also allowed on the right side

```

1 class Solution:
2     #Back-end complete function Template for Python 3
3
4     #Function to find the leaders in the array.
5     def leaders(self, n, arr):
6         #Code here
7         max = float('-inf')
8         result_list = []
9         for i in range(n-1,-1,-1):
10             if max <= arr[i]:
11                 result_list.append(arr[i])
12                 max = arr[i]
13         return result_list[::-1]
14
15
16

```

### Code:

```

def leaders(n, arr):
    #Code here
    max = float('-inf')
    result_list = []
    for i in range(n-1,-1,-1):
        if max <= arr[i]:
            result_list.append(arr[i])
            max = arr[i]
    return result_list

```

### Example:

```
leaders(6,[16,17,4,3,5,2]) # Output: [2, 5, 17]
```

## 2.5.3 Problem 3: Sort 0s, 1s and 2s

### Problem Statement([Link](#)):

Given an array `arr` containing only **0s, 1s, and 2s**. Sort the array in ascending order. **Expected Time Complexity:**  $O(n)$

### Challenges Faced:

- A brute force approach such as using sorting algorithms like quicksort, mergesort, or Python's built-in sorting function could lead to a time complexity of  $O(n \log n)$ . The challenge is to sort the array in linear time and with constant space.
- Sorting an array with three distinct values (0s, 1s, and 2s) requires a strategy to efficiently place them in their correct positions without using extra space.
- A simple approach like counting the frequency of each element (0, 1, and 2) and rearranging them afterward works but still requires two passes over the array, which is not optimal.

- The requirement to sort the array in-place (without additional space) adds complexity. Swapping the elements based on their value (0, 1, or 2) must be done in such a way that it maintains the overall structure and correctly positions all elements.

#### Approach:

- The optimal approach to solve this problem in  $O(n)$  time and  $O(1)$  space is the **Dutch National Flag Algorithm**. All the elements which have the value 0 will be on the left, the ones which have the value 1 will be in the middle and the ones with value as 2 will be on the right.
- Three pointer technique is used to solve this problem. The first pointer(*low*) and second pointer(*mid*) will first point to the leftmost element and the third pointer(*high*) will point to the rightmost element.
- Everything to the left of *low* will be 0. *Mid* pointer moves through the array comparing elements and determining whether to swap or not. Everything to the right of *high* will be 2.
- If  $\text{arr}[\text{mid}] == 0$ , the element in  $\text{arr}[\text{mid}]$  will be swapped with  $\text{arr}[\text{low}]$  (placing the 0 in its correct position) and the value of both *low* and *mid* will be incremented by 1.
- If  $\text{arr}[\text{mid}] == 1$ , *mid* pointer is incremented (since 1 is already in its correct position).
- If  $\text{arr}[\text{mid}] == 2$ ,  $\text{arr}[\text{mid}]$  is swapped with  $\text{arr}[\text{high}]$  (placing the 2 in its correct position), and the *high* pointer will be decremented. However, *mid* pointer will not be incremented because the swapped element from the *high* position needs to be checked again.
- The process is continued until *mid* exceeds *high*, ensuring that all elements are in their correct sections.
- If the array is already sorted, the algorithm still works and performs only minimal operations as the *mid* pointer traverses the array and if the array contains only one type of element (e.g., all 0s or all 1s), the algorithm still runs, but no swaps are needed, so it performs efficiently.

#### Solution:

**Sort 0s, 1s and 2s**

Difficulty: Easy Accuracy: 50.58% Submissions: 695K+ Points: 2

Given an array `arr` containing only 0s, 1s, and 2s. Sort the array in ascending order.

Examples:

Input: `arr[] = [0, 2, 1, 2, 0]`

Output: `0 0 1 2 2`

Explanation: 0s 1s and 2s are segregated into ascending order.

Input: `arr[] = [0, 1, 0]`

Output: `0 0 1`

Explanation: 0s 1s and 2s are segregated into ascending order.

Expected Time Complexity:  $O(n)$

Expected Auxiliary Space:  $O(1)$

```

1 class Solution:
2     # Function to sort an array of 0s, 1s, and 2s
3     def sort012(self, arr):
4         # code here
5         n = len(arr)
6         low = 0
7         mid = 0
8         high = n-1
9
10        while mid <= high:
11            if arr[mid] == 2:
12                arr[mid], arr[high] = arr[high], arr[mid]
13                high -= 1
14            elif arr[mid] == 0:
15                arr[low], arr[mid] = arr[mid], arr[low]
16                low += 1
17                mid += 1
18            elif arr[mid] == 1:
19                mid += 1
20        return arr
21
22
23

```

#### Code:

```

def sort012(arr):
    n = len(arr)
    low = 0
    mid = 0
    high = n-1

    while mid <= high:
        if arr[mid] == 2:
            arr[mid], arr[high] = arr[high], arr[mid]
            high -= 1
        elif arr[mid] == 0:
            arr[low], arr[mid] = arr[mid], arr[low]
            low += 1
            mid += 1

```

```
elif arr[mid] == 1:
    mid += 1
return arr
```

Example:

```
sort012([0, 2, 1, 2, 0]) # Output: [0, 0, 1, 2, 2]
```

#### 2.5.4 Problem 4: Count more than $n/k$ occurrences

##### Problem Statement([Link](#)):

Given an array **arr** of size **N** and an element **k**. The task is to find the count of elements in the array that appear more than  $n/k$  times.

##### Challenges Faced:

- The main challenge was determining how often each element appears in the array. This required maintaining an efficient way to store the count of each element as the array is iterated.
- When working with large datasets, maintaining efficient time complexity while checking for the occurrence of each element in the array was crucial. Without an efficient approach, counting each element multiple times could lead to performance issues.
- The challenge of avoiding repeated element checks was crucial. A brute-force approach could involve multiple scans of the array to count element frequencies, which would result in inefficient time complexity (e.g.,  $O(n^2)$ ).
- There was a need to ensure that unique elements were handled correctly without counting them more than once.

##### Approach:

- To count how many times each element appears in the array, a dictionary (candidates) is used. This allows for efficient lookups and updates, both of which happen in constant time ( $O(1)$ ) on average.
- As the array is traversed, each element is checked in the dictionary. If it already exists, its count is incremented; otherwise, it is added with a count of 1. This ensures that each element is counted only once during the array traversal.
- Once the frequency of each element is calculated, the dictionary is iterated through to find how many elements appear more than  $n // k$  times.
- The threshold  $n // k$  is calculated, and for each element in the dictionary, its count is compared to this threshold. If it exceeds the threshold, that element is considered a valid result.
- By using a dictionary, the solution avoids repeated scans of the array and ensures that counting and lookup operations are done in constant time, resulting in a linear time complexity of  $O(n)$ .

##### Solution:

### Count More than n/k Occurrences

Difficulty: Easy Accuracy: 58.35% Submissions: 111K+ Points: 2

Given an array `arr` of size `N` and an element `k`. The task is to find the count of elements in the array that appear more than  $n/k$  times.

#### Example 1:

##### Input:

`N = 8`  
`arr = [3,1,2,2,1,2,3,3]`  
`k = 4`

##### Output:

2

##### Explanation:

In the given array, 3 and 2 are the only elements that appears more than  $n/k$  times.

#### Example 2:

##### Input:

`N = 4`  
`arr = [2,3,3,2]`  
`k = 3`

##### Output:

2

**Explanation:** In the given array, 3 and 2 are the only elements that appears more than  $n/k$  times. So the count of elements are 2.

#### Your Task:

The task is to complete the function `countOccurrence()` which returns count of elements with more than  $n/k$  times appearance.

```
1 #User function template for python
2
3
4 class Solution:
5
6     #Function to find all elements in array that appear more than n/k times.
7     def countOccurrence(self,arr,n,k):
8         #Your code here
9         candidates = {}
10
11         for num in arr:
12             if num in candidates:
13                 candidates[num] += 1
14             else:
15                 candidates[num] = 1
16
17
18         result = 0
19         for candidate, count in candidates.items():
20             if count > n // k:
21                 result += 1
22
23         return result
24
25
26
27
```

Code:

```
def countOccurrence(arr,n,k):
    candidates = {}

    for num in arr:
        if num in candidates:
            candidates[num] += 1
        else:
            candidates[num] = 1

    result = 0
    for candidate, count in candidates.items():
        if count > n // k:
            result += 1

    return result
```

Example:

`arr = [3,1,2,2,1,2,3,3]`

`n = 8`

`k = 4`

```
countOccurence(arr,n,k) # Output: 2
```

### 2.5.5 Problem 5: Kadane's Algorithm

#### Problem Statement([Link](#)):

Given an integer array **arr[]**. Find the contiguous sub-array(containing at least one number) that has the maximum sum and return its sum.

#### Challenges Faced:

- Kadane's Algorithm needs to properly handle arrays where all elements are negative. The challenge is ensuring that the algorithm correctly identifies the maximum sum, which will be the largest single negative number in such cases.
- The algorithm must efficiently manage the sum of the current subarray. Resetting the subarray sum to zero when it becomes negative is crucial to avoid unnecessary calculations and to ensure that only potential maximum subarrays are considered.
- Maintaining and updating the maximum sum encountered so far while traversing the array requires careful comparison to ensure that the global maximum sum is correctly identified.
- The algorithm must handle edge cases such as: Arrays with a single element, Arrays where all elements are negative, and Large arrays to ensure that the solution remains efficient and within acceptable time limits

#### Approach:

- max\_sum is initialized to the smallest possible value (-infinity). This ensures that any value in the array will be larger initially.
- current\_sum is initialized to 0. This keeps track of the sum of the current subarray.
- The array is traversed and for each element, the current\_sum is updated by adding the current element to it.
- The next step is to check if current\_sum is greater than max\_sum and update max\_sum if necessary.
- If current\_sum becomes negative, it is reset to 0. This is because a negative current\_sum would not contribute positively to future subarrays and would only reduce the potential sum.
- After completing the iteration through the array, max\_sum will contain the maximum sum of a contiguous subarray.
- This solution handles the edge cases as well. If the array has single element, the loop will run only one time and the max sum will be the current element.
- If all elements in the array are negative, current\_sum will consistently be negative after each addition. Therefore, the algorithm will continuously reset current\_sum to 0. Because of the resetting of current\_sum, max\_sum will end up being the largest single negative element in the array. This is because when all subarray sums are negative, the largest value among them is simply the least negative number.

#### Solution:

#### Kadane's Algorithm

Difficulty: Medium Accuracy: 36.28% Submissions: 974K+ Points: 4

Given an integer array **arr[]**. Find the contiguous sub-array(containing at least one number) that has the maximum sum and return its sum.

**Examples:**

**Input:** arr[] = [1, 2, 3, -2, 5]  
**Output:** 9  
**Explanation:** Max subarray sum is 9 of elements (1, 2, 3, -2, 5) which is a contiguous subarray.

**Input:** arr[] = [-1, -2, -3, -4]  
**Output:** -1  
**Explanation:** Max subarray sum is -1 of element (-1)

**Input:** arr[] = [5, 4, 7]  
**Output:** 16  
**Explanation:** Max subarray sum is 16 of element (5, 4, 7)

**Expected Time Complexity:** O(n)  
**Expected Auxiliary Space:** O(1)

```
1 #User function Template for python3
2
3 class Solution:
4     #Complete this function
5     #Function to find the sum of contiguous subarray with maximum sum.
6     def maxSubArraySum(self, arr):
7         #Your code here
8         max_sum = float('-inf')
9         curr_sum = 0
10
11         for i in arr:
12             curr_sum += i
13
14             if curr_sum > max_sum:
15                 max_sum = curr_sum
16
17             if curr_sum < 0:
18                 curr_sum = 0
19
20         return max_sum
21
22
23
```

Code:

```
def maxSubArraySum(arr):  
    ##Your code here  
    max_sum = float('-inf')  
    curr_sum = 0  
  
    for i in arr:  
        curr_sum += i  
  
        if curr_sum > max_sum:  
            max_sum = curr_sum  
  
        if curr_sum < 0:  
            curr_sum = 0  
  
    return max_sum
```

Example:

```
arr = [1, 2, 3, -2, 5]  
maxSubArraySum(arr)    # Output: 9
```

### 2.5.6 Problem 6: Smallest Positive Missing Number

#### Problem Statement([Link](#)):

You are given an array **arr[]** of **N** integers. The task is to find the smallest positive number missing from the array.

#### Challenges Faced:

- The Brute Force approach involves keeping track of all the positive numbers and checking if all of them are present in the array. This approach will require nested loop and will have the time complexity of  $O(n^2)$ .
- Brute Force code:

```
def missingNumber(arr,n):  
    check_present = False  
    for i in range(1, n+1):  
        check_present = False  
        for j in range(n):  
            if arr[j] == i:  
                check_present = True  
                break  
        if check_present == False:  
            return i
```

```
return i+1
```

- The input array can contain negative numbers and zeros, which are not relevant for finding the smallest positive missing number.
- The solution needs to efficiently handle large arrays, ensuring that the time complexity is acceptable given the constraints.
- The input array might have duplicate elements, which should be managed correctly to avoid unnecessary checks.
- The algorithm must not use extra space, so it rearranges the array in-place to bring all positive numbers to the left. This adds complexity since the array's original structure is altered during execution.
- The algorithm changes the sign of elements to mark the presence of a number. If an element is already negative, care must be taken to prevent double marking or skipping.
- After marking, the algorithm must correctly identify the first index where the value is still positive, which represents the missing positive integer. If all indices are marked, the next number ( $n + 1$ ) is returned as the missing number.

#### Approach 1:

- The array is converted into a hash set. This allows for  $O(1)$  average time complexity checks for the presence of integers. This handles the presence of negative numbers and zero by ignoring them, as only positive numbers are considered while iterating.
- The hash set is iterated from 1 up to  $n + 1$  (where  $n$  is the length of the array). This range is sufficient because if the array contains all positive integers from 1 to  $n$ , then the smallest missing positive integer must be  $n + 1$ .
- For each number in the range  $[1, n + 1]$ , it is checked if it exists in the hash set. The first number not found in the set is the smallest missing positive integer.

#### Approach 2:

- Here the indexes are used to keep track if all the positive numbers are present or not. Since the positive numbers start from 1, for an array of size  $n$ , if all the positive numbers are present that means the numbers are from 1 to  $n$  and the next smallest positive number not present in the array will be  $n+1$ .
- If an element is present in the array then the actual element in that index will be marked as negative which will let us know in the end if there is any missing positive element.
- The function first moves all positive numbers to the left side of the array while keeping the non-positive numbers on the right. The index  $j$  tracks the number of positive elements. This allows the algorithm to focus on the first  $j$  elements, which are the relevant positive numbers.
- The algorithm iterates through the first  $j$  elements (the positive numbers) and for each value  $val$ , marks the corresponding index  $val - 1$  as negative. This indicates that the number  $val$  is present in the array. The absolute value is taken to ensure that previously marked numbers do not interfere with marking other numbers.
- After marking, the algorithm iterates through the first  $j$  elements again. The first positive value indicates the smallest missing positive number (i.e., index + 1). If all elements are marked, meaning all numbers from 1 to  $j$  are present, the missing number is  $j + 1$ .

#### Solution:

**Approach 1:** Time Complexity  $O(n)$ , Space Complexity  $O(n)$



### Smallest Positive Missing

Difficulty: Medium Accuracy: 25.13% Submissions: 341K+ Points: 4

You are given an array `arr[]` of `N` integers. The task is to find the smallest positive number missing from the array.

**Note:** Positive number starts from 1.

**Example 1:**

**Input:**

`N = 5`  
`arr[] = {1,2,3,4,5}`

**Output:** 6

**Explanation:** Smallest positive missing number is 6.

**Example 2:**

**Input:**

`N = 5`  
`arr[] = {0,-10,1,3,-20}`

**Output:** 2

**Explanation:** Smallest positive missing number is 2.

**Your Task:**  
The task is to complete the function `missingNumber()` which returns the smallest positive missing number in the array.

```

1  #User function Template for python3
2
3  class Solution:
4
5      #Function to find the smallest positive number missing from the array.
6      def missingNumber(self,arr,n):
7          #Your code here
8          hash = set(arr)
9          for i in range(1, n + 2):
10             if i not in hash:
11                 return i
12
13
14  # } Driver Code Ends

```

Code:

```

def missingNumber(arr,n):

    hash = set(arr)

    for i in range(1, n + 2):

        if i not in hash:

            return i

```

Example:

```

arr = [0,-10,1,3,-20]

n = len(arr)

missingNumber(arr,n)  # Output: 2

```

**Approach 2:** Time Complexity  $O(n)$ , Space Complexity  $O(1)$

### Smallest Positive Missing

Difficulty: Medium Accuracy: 25.13% Submissions: 341K+ Points: 4

You are given an array `arr[]` of `N` integers. The task is to find the smallest positive number missing from the array.

**Note:** Positive number starts from 1.

**Example 1:**

**Input:**

`N = 5`  
`arr[] = {1,2,3,4,5}`

**Output:** 6

**Explanation:** Smallest positive missing number is 6.

**Example 2:**

**Input:**

`N = 5`  
`arr[] = {0,-10,1,3,-20}`

**Output:** 2

**Explanation:** Smallest positive missing number is 2.

**Your Task:**  
The task is to complete the function `missingNumber()` which returns the smallest positive missing number in the array.

```

1  #User function Template for python3
2
3  class Solution:
4
5      #Function to find the smallest positive number missing from the array.
6      def missingNumber(self,arr,n):
7          #Your code here
8          j = 0
9          for i in range(n):
10             if arr[i] > 0:
11                 arr[i], arr[j] = arr[j], arr[i]
12                 j += 1
13
14             for k in range(j):
15                 val = abs(arr[k])
16
17                 if val - 1 < j and arr[val - 1] > 0:
18                     arr[val - 1] *= -1
19
20             for l in range(j):
21                 if arr[l] > 0:
22                     return l+1
23
24             return j + 1
25
26
27

```

Code:

```
def missingNumber(arr,n):  
    # move negative numbers to the right side and positive numbers to the left side  
  
    j = 0  
    for i in range(n):  
        if arr[i] > 0:  
            arr[i], arr[j] = arr[j], arr[i]  
            j += 1  
  
    for k in range(j):  
        val = abs(arr[k])  
  
        if val - 1 < j and arr[val - 1] > 0:  
            arr[val - 1] *= -1  
  
    for l in range(j):  
        if arr[l] > 0:  
            return l+1  
  
    return j + 1
```

Example:

```
arr = [3,4,-1,1]  
n = len(arr)  
missingNumber(arr,n) # Output: 2
```

## 2.6 Key Takeaways from the Resources

- Arrays are essential when dealing with indexed data where fast access to elements is required.
- Contiguous memory allocation provides speed advantages but can also lead to memory inefficiencies when working with dynamic data.
- Arrays are frequently used in various applications, such as storing matrices, buffers in operating systems, and lookup tables in databases.
- However, their inability to dynamically resize and the overhead of insertion and deletion are significant drawbacks.
- Common alternatives, like linked lists, address some of these limitations but come at the cost of losing constant-time element access.

## 3. Linked Lists

### 3.1 Introduction to Linked Lists

**Key Concept:** A Linked List is a linear data structure where elements are stored in nodes, and each node contains two parts:

1. **Data:** The actual value of the node.
2. **Pointer/Reference:** A reference to the next node in the sequence.

Unlike arrays, linked lists don't store elements in contiguous memory locations. Instead, they are dynamically allocated, allowing them to grow or shrink in size during runtime. The last node has a pointer that points to NULL, indicating the end of the list. The head node marks the starting point of the linked list.

There are different types of linked lists:

- **Singly Linked List:** Each node points to the next one.
- **Doubly Linked List:** Each node points to both the next and the previous node.
- **Circular Linked List:** The last node points back to the head, creating a circular structure.

### 3.2 Memory Allocation

In a **linked list**, memory is allocated dynamically as needed. This means that the list can grow or shrink in size based on the operations performed, unlike arrays where memory allocation is fixed at the time of creation.

- Each node in a linked list is stored in a separate memory location, and nodes are not stored contiguously. This provides flexibility in terms of size but adds overhead due to the need to store pointers/references to other nodes.
- Since each node stores a reference to another node (in singly or doubly linked lists), there is an additional memory overhead for pointers, which is not required in arrays.

### 3.3 Applications, Benefits and Limitations of Linked Lists

#### Applications:

- Linked lists form the foundation for other data structures like stacks, queues, and graphs.
- Useful when the number of elements is not known in advance or may change during the execution of a program.
- Linked lists are used in applications like LRU cache, memory management in operating systems, and process scheduling.
- Linked lists are useful in arithmetic operations on long integers and in algorithms that need to frequently insert or delete items from large collections of data.

#### Benefits:

- Unlike arrays, linked lists can grow or shrink during execution, which makes them ideal for use cases where the number of elements is not known beforehand.
- Insertions and deletions can be done efficiently, especially at the beginning or middle of the list, without needing to shift elements as in an array.
- Memory is allocated only when needed, avoiding wastage associated with pre-allocated memory in arrays.
- Implementation of Abstract Data Types: Linked lists are the foundation for abstract data types like stacks, queues, and hash tables.

#### Limitations:

- Accessing elements in a linked list requires traversing from the head, making access times slower ( $O(n)$ ) compared to arrays, which have constant-time access ( $O(1)$ ).
- Each node requires extra memory for the pointer/reference, which makes linked lists less memory-efficient compared to arrays.
- Linked lists exhibit poor cache locality due to their non-contiguous memory allocation. Arrays, with contiguous memory, benefit from better cache performance.
- In singly linked lists, reverse traversal is not possible unless additional data structures or modifications (like doubly linked lists) are used, adding to complexity.

### 3.4 Linked List Vs Arrays

Both linked lists and arrays are fundamental data structures, but they have distinct characteristics in terms of performance, memory usage, and use cases.

- Linked lists have dynamic memory allocation where nodes are allocated as needed and arrays have static memory allocation where the memory size is predefined.
- Linked lists use more memory as they also store the pointers, which is extra space for prev/next addresses as compared to arrays, which do not store any pointers.
- Arrays need contiguous memory space in the memory which linked lists do not need as the next nodes could be stored anywhere, just the address of the node should be stored in the pointer of previous node.
- Array has good cache performance as it uses contiguous memory and it is poor in linked list as it is stored in non contiguous memory locations.
- Traversing takes  $O(n)$  time in linked list as traversing is done from head and it is  $O(1)$  in array as the element can be directly accessed in arrays.
- In linked list insertion of element at the beginning only takes  $O(1)$  time as a new element has to be made head which will point to the previous head and it is  $O(n)$  time complexity to enter values at the middle or end as the linked list has to be traversed to insert an element. While in arrays, it takes  $O(n)$  time to insert an element as the other elements has to be shifted after the new element is inserted into place.
- For deletion, it is the same case where it takes  $O(1)$  to delete elements from the start of the linked list and  $O(n)$  to delete from the middle or end and in arrays it takes  $O(n)$  time complexity to delete and element as the other elements has to be shifted after deletion.
- Linked lists are efficient for dynamic data structures where size changes and arrays are efficient for fixed-size data structures.

### 3.5 Basic Operations

Basic operations performed in a linked list are creation, insertion, deletion, traversal and searching.

#### 1. Creation:

To create a linked list, we first need to define a Node class that holds the data and a reference to the next node.

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

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

Here, the Node class defines each element in the list, and the LinkedList class initializes the list with a head pointing to None.

#### 2. Insertion:

Insertion at the beginning:

```
def insert_at_beginning(self, value):
    current_head = self.head
    new_node = ListNode(value)
    new_node.next = current_head
    self.head = new_node
```

Insert at end:

```
def insert_at_end(self, value):
```

```

if len(self) == 0:
    new_node = ListNode(value)
    self.head = new_node
else:
    current_node = self.head
    while current_node:
        if not current_node.next:
            current_tail = current_node
            new_node = ListNode(value)
            current_tail.next = new_node
            new_node.next = None
            break
        else:
            current_node = current_node.next

```

Insert at index:

```

def insert_at_index(self, index, value):
    current_node = self.head
    current_node_index = 0
    if index == 0:
        new_node = ListNode(value)
        new_node.next = current_node
        self.head = new_node
    elif 1 <= index < len(self):
        while current_node:
            if current_node_index + 1 == index:
                new_node = ListNode(value)
                next_node = current_node.next
                current_node.next = new_node
                new_node.next = next_node
                break
            else:
                current_node = current_node.next
                current_node_index += 1

```

### 3. Deletion:

Delete from beginning:

```

def delete_from_beginning(self):
    self.head = self.head.next

```

Delete from end:

```

def delete_from_end(self):

```

```

current_node = self.head
while current_node:
    if current_node.next.next == None:
        current_node.next = None
        break
    else:
        current_node = current_node.next

```

Delete value:

```

def delete_value(self, value):
    current_node = self.head
    previous_node = None
    while current_node:
        if current_node.value == value:
            if previous_node == None:
                self.head = self.head.next
            else:
                previous_node.next = current_node.next
            return current_node.value
        else:
            previous_node = current_node
            current_node = current_node.next
    return -1

```

#### 4. Traversal:

```

def __repr__(self):
    values = []
    current_node = self.head
    while current_node:
        values.append(str(current_node.value))
        current_node = current_node.next
    if values:
        values.append('None')
        return ' -> '.join(values)
    else:
        return 'Linked list is empty'

```

#### 5. Searching:

```

def search_value(self, value):
    current_node = self.head
    current_node_index = 0

```

```

while current_node:
    if current_node.value == value:
        return current_node_index
    else:
        current_node = current_node.next
        current_node_index += 1
return -1

```

## 3.6 Hands-on Coding Exercises

### 3.6.1 Problem 1: Reverse Linked List

#### Problem Statement([Link](#)):

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

#### Challenges Faced:

- While reversing the singly linked list it is possible to lose a node, which is why three pointers are used in the solution so that one of them temporarily stores the next node before changing the current.next pointer.
- Handling edge cases like an empty list or a list with only one node can be tricky. For an empty list, the head is None, and for a single-node list, the head should remain unchanged. To handle this, prev is initialized to None and the while loop gets terminated when current becomes None..
- It is difficult to correctly reverse the list while using minimal space. The iterative approach is used here, which operates in  $O(1)$  space complexity by using a fixed number of pointers (prev, current, next\_node). This avoids additional space and ensures optimal performance.
- The loop must correctly terminate when the end of the list is reached. Incorrectly managing the termination condition can lead to infinite loops or incorrect results. The while loop continues as long as current is not None. This ensures that each node is processed, and the loop exits correctly when the end of the list is reached.

#### Approach:

- Previous node(prev\_node) is initialized to None because the reversed list will end with None. Current node(current\_node) starts at the head of the list and next node(next\_node) is also initialized to None and changed later in the while loop.
- The while loop continues as long as current\_node is not None.
- next\_node temporarily stores the next node to avoid losing it when reversing the link.
- current\_node.next is set to prev\_node to reverse the direction.
- Prev\_node is updated to current\_node
- current\_node is updated to next\_node to continue traversing the list. to move one step forward.
- Once the loop ends, prev\_node points to the new head of the reversed linked list. This is returned as the result.

#### Time Complexity:

- $O(n)$  as each node is processed exactly once, where n is the number of nodes in the linked list.

#### Solution:

## 206. Reverse Linked List

Easy Topics Companies

Given the `head` of a singly linked list, reverse the list, and return the reversed list.

**Example 1:**

```

Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]

```

**Example 2:**

```

Input: head = [1,2]
Output: [2,1]

```

```

1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, val=0, next=None):
4 #         self.val = val
5 #         self.next = next
6 class Solution:
7     def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
8         current_node = head
9         previous_node = None
10        next_node = None
11        while current_node:
12            next_node = current_node.next
13            current_node.next = previous_node
14            previous_node = current_node
15            current_node = next_node
16
17        return previous_node

```

Testcase Test Result

Accepted Runtime: 46 ms

Case 1 Case 2 Case 3

Input

head =

[1,2,3,4,5]

Code:

```

def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:

    current_node = head
    previous_node = None
    next_node = None
    while current_node:
        next_node = current_node.next
        current_node.next = previous_node
        previous_node = current_node
        current_node = next_node

    return previous_node

```

Example:

```

head = [1,2,3,4,5]
reverseList(head)

# Output: [5,4,3,2,1]

```

### 3.6.2 Problem 2: Middle of the Linked List

#### Problem Statement([Link](#)):

Given the head of a singly linked list, return *the middle node of the linked list*.  
If there are two middle nodes, return **the second middle** node.

#### Challenges Faced:

- In an odd-length list, the middle node is straightforward, but in an even-length list correctly identifying the middle node when there are two middle nodes is required. The approach naturally handles both cases. When fast reaches the end of the list in an even-length list, slow will be at the second middle node due to the two-pointer technique.



- Finding the middle node in a single pass through the list is crucial for efficiency. The two-pointer technique ensures that you only traverse the list once, making the solution efficient.
- Handling cases where the list is empty or has only one node is challenging. The solution handles these cases naturally. If the list is empty (head is None), slow will also be None. If there's only one node, slow will correctly point to this single node as it becomes the middle node.

### Approach:

- Two pointers 'slow' and 'fast' is initialized. 'slow' pointer starts at the head of the list and will move one node at a time. 'fast' pointer also starts at the head of the list but moves two nodes at a time.
- The list is traversed while fast and fast.next are not None. In each step, 'slow' moves one node forward and 'fast' moves two nodes forward.
- When 'fast' reaches the end of the list, 'slow' will be at the middle node. After the loop ends, 'slow' will point to the middle node of the list.

### Time Complexity:

- $O(n)$  as the list is traversed only once by the fast and slow pointers, where  $n$  is the number of nodes in the list.

### Solution:


#### 876. Middle of the Linked List

Easy Topics Companies

Given the `head` of a singly linked list, return the *middle node* of the linked list.


If there are two middle nodes, return the *second middle node*.

**Example 1:**



Input: head = [1,2,3,4,5]  
Output: [3,4,5]  
Explanation: The middle node of the list is node 3.

**Example 2:**



Input: head = [1,2,3,4,5,6]  
Output: [4,5,6]  
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.

**Constraints:**

- The number of nodes in the list is in the range [1, 100].
- $1 \leq \text{Node.val} \leq 100$

```

1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, val=0, next=None):
4 #         self.val = val
5 #         self.next = next
6 class Solution:
7     def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
8         slow = head
9         fast = head
10        while slow and fast and fast.next:
11            slow = slow.next
12            fast = fast.next.next
13        return slow
14

```

Saved Ln 1, Col 1

Testcase Test Result

Accepted Runtime: 59 ms

Case 1 Case 2

Input

### Code:

```

def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
    slow = head
    fast = head
    while slow and fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

```

### Example:

```

head = [1,2,3,4,5]
middleNode(head)

#Output: [3,4,5]

```

### 3.7 Key Takeaways from the Resources

- Linked lists allocate memory dynamically, allowing for flexible sizing and efficient insertions/deletions without shifting elements like arrays.
- Linked lists have slower access times ( $O(n)$ ) due to sequential traversal, unlike arrays which have constant-time access ( $O(1)$ ).
- Extra memory is required for pointers/references in each node, making linked lists less memory-efficient than arrays.
- Linked Lists are ideal for applications where frequent insertions/deletions occur, but less suitable for scenarios demanding fast element access.

## 4. Stacks

### 4.1 Introduction to Stacks

**Key Concept:** A stack is a linear data structure that follows the **LIFO (Last In First Out)** principle, meaning the last element added to the stack is the first one to be removed. Stacks are used in various applications, such as expression evaluation, backtracking algorithms, and managing function calls in recursive algorithms.

**Structure:**

- **Top:** Refers to the topmost element in the stack. It's the only element that can be accessed or removed.
- **Push:** Adds an element to the top of the stack.
- **Pop:** Removes the element at the top of the stack.

### 4.2 Memory Allocation

Stacks use contiguous memory allocation in the form of arrays or dynamic memory allocation when implemented using linked lists.

- In array-based stacks, the size of the stack is fixed, and memory is allocated at once.
- In linked list-based stacks, memory is allocated dynamically, and the size of the stack can grow or shrink.

### 4.3 Applications, Benefits and Limitations of Stacks

**Applications:**

- Stacks are used to evaluate expressions like postfix, prefix, and infix.
- Stacks manage function calls during recursion, ensuring functions are executed in the correct order.
- Algorithms like DFS (Depth First Search) use stacks for exploring possible solutions.
- Stacks are used in editors to reverse operations or retrieve previous states.

**Benefits:**

- Stacks allow for quick and efficient insertion and deletion operations (push and pop).
- Stacks use only the required memory when using linked lists and can dynamically resize.
- Stacks are suited for scenarios where the most recent action or value needs to be accessed first.

**Limitations:**

- Only the top element of the stack can be accessed, making random access to elements difficult.
- For static arrays, the stack has a fixed size, limiting its flexibility.
- In fixed-size stacks, stack overflow is risky if too many elements are added, and stack underflow is risky when removed from an empty stack.

### 4.4 Basic Operations

Basic operations performed on a Stack are:

- **Push Operation:** Add an element to the top of the stack.
- **Pop Operation:** Remove the element at the top of the stack.
- **Peek Operation:** Retrieve the top element without removing it.
- **IsEmpty Operation:** Check if the stack is empty.
- **IsFull Operation:** Check if the stack is full (for array-based stacks).

```
class Stack:
    def __init__(self, capacity):
        self.stack = []
        self.capacity = capacity

    # Push operation
    def push(self, value):
```

```

    if len(self.stack) < self.capacity:
        self.stack.append(value)
    else:
        print("Stack Overflow")

# Pop operation
def pop(self):
    if not self.is_empty():
        return self.stack.pop()
    else:
        print("Stack Underflow")
        return None

# Peek operation
def peek(self):
    if not self.is_empty():
        return self.stack[-1]
    else:
        return None

# IsEmpty operation
def is_empty(self):
    return len(self.stack) == 0

# IsFull operation (for array-based stacks)
def is_full(self):
    return len(self.stack) == self.capacity

# Example usage
stack = Stack(5)
stack.push(1)
stack.push(2)
print(stack.pop()) # Outputs: 2
print(stack.peek()) # Outputs: 1

```

## 4.5 Hands-on Coding Exercises

### 4.5.1 Problem 1: Remove All Adjacent Duplicates in String

**Problem Statement**([Link](#)):

You are given a string *s* consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

We repeatedly make duplicate removals on *s* until we no longer can.

Return *the final string after all such duplicate removals have been made*. It can be proven that the answer is unique.

### Challenges Faced:

- Identifying and removing adjacent duplicates efficiently can be tricky, especially when the string might require multiple passes to fully remove all duplicates. The stack-based approach solves this by processing each character and using the stack to manage the removal of adjacent duplicates in a single pass.
- After removing duplicates, the order of the remaining characters must be preserved. The stack approach maintains this order naturally by appending characters in sequence and removing only the adjacent duplicates. The stack preserves the order of characters as they are processed.
- Handling edge cases such as an empty string or a string with no adjacent duplicates is a challenge. The solution naturally handles these cases. An empty string will return as an empty string, and strings with no adjacent duplicates will remain unchanged.

### Approach:

- The solution uses a stack-based approach to efficiently handle adjacent duplicate removals. A stack is initialized to keep track of characters as we process the string.
- It is checked if each character in the string matches the character at the top of the stack (i.e., the last added character).
- If it matches, it means adjacent duplicates have been found, so the top of the stack is removed (pop operation).
- If it doesn't match, the character is added to the stack (append operation).
- The stack will contain the final string with all adjacent duplicates removed. In the end, the stack is converted to a string and is returned.

### Why Stack?

- The stack data structure is ideal for problems involving consecutive processing and removal. It allows for efficient access to the last added element (top of the stack), which is essential for identifying and removing adjacent duplicates.
- By iterating through the string once and using the stack to manage the removals, the approach ensures a linear time complexity, making it suitable for handling large strings.

### Time Complexity:

- $O(n)$ . The solution processes each character in the string exactly once. Each character is pushed onto the stack or removed from it at most once, resulting in a linear time complexity relative to the length of the string.

### Solution:

The screenshot shows the LeetCode interface for problem 1047. On the left, the problem description states: "You are given a string *s* consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them. We repeatedly make duplicate removals on *s* until we no longer can. Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique." Examples provided are: Input: "abbaca", Output: "ca"; and Input: "azxxzy", Output: "ay". On the right, the Python solution is shown in a code editor. It defines a class `Solution` with a method `removeDuplicates` that uses a stack to process the string. The test result at the bottom shows "Accepted" with a runtime of 51 ms.

```
1 class Solution:
2     def removeDuplicates(self, s: str) -> str:
3         stack = []
4
5         for char in s:
6             if stack and stack[-1] == char:
7                 stack.pop()
8             else:
9                 stack.append(char)
10
11        return ''.join(stack)
```

Code:

```
def removeDuplicates(s):
```

```

stack = []

for char in s:
    if stack and stack[-1] == char:
        stack.pop()
    else:
        stack.append(char)

return ''.join(stack)

```

Example:

```

s = "abbaca"
removeDuplicates(s)

# Output: 'ca'

```

#### 4.5.2 Problem 2: Min Stack

##### Problem Statement([Link](#)):

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

##### Challenges Faced:

- The challenge is to design a data structure to support push, pop, top, and getMin in constant time while ensuring correctness. A secondary stack is used to track the minimum values allowing all operations to be performed in constant time.
- Next challenge is ensuring that operations like pop and getMin handle cases where the stack might be empty or when multiple elements with the same minimum value are present. Properly managing the secondary stack (min\_stack) ensures that these edge cases are handled correctly.
- It has to be ensured that main\_stack and min\_stack remain in sync, especially when popping elements. The solution ensures that when the top element of main\_stack is removed, it also removes the element from min\_stack if it matches the top of min\_stack.

##### Approach:

- The solution uses two stacks to achieve the required O(1) time complexity for each operation:
  - **Main Stack:** Stores all the elements in the stack.
  - **Min Stack:** Keeps track of the minimum elements.
- `__init__`: Initializes two stacks, main\_stack for storing elements and min\_stack for keeping track of the minimum values.
- **Push:** Adds the value to main\_stack and updates min\_stack if the new value is smaller than or equal to the current minimum.
- **Pop:** Removes the top element from main\_stack and also removes the top element from min\_stack if it matches the value being removed.
- **Top:** Returns the top element from main\_stack.

- **getMin**: Returns the top element from `min_stack`, which is the current minimum value.
- The minimum element can be retrieved in a constant time by using a separate stack (`min_stack`) to store the minimum values. The `min_stack` only updates when a new minimum is pushed onto it.
- Both push and pop operations are managed in  $O(1)$  time. When an element is pushed, it's added to both `main_stack` and `min_stack` if it's less than or equal to the current minimum. When an element is popped, it's removed from both stacks if it matches the current minimum.

### Time Complexity:

- **push(val)**:  $O(1)$  - Pushing onto `main_stack` and conditionally onto `min_stack` takes constant time.
- **pop()**:  $O(1)$  - Popping from `main_stack` and conditionally from `min_stack` takes constant time.
- **top()**:  $O(1)$  - Accessing the top element of `main_stack` takes constant time.
- **getMin()**:  $O(1)$  - Accessing the top element of `min_stack` takes constant time.

### Solution:

#### 155. Min Stack

Medium Topics Companies Hint

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with  $O(1)$  time complexity for each function.

**Example 1:**

**Input**  
["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]  
[[], [-2], [0], [-3], [], [], [], []]

**Output**  
[null, null, null, null, -3, null, 0, -2]

**Explanation**  
MinStack minStack = new MinStack();  
minStack.push(-2);  
minStack.push(0);  
minStack.push(-3);  
minStack.getMin(); // return -3  
minStack.pop();  
minStack.top(); // return 0  
minStack.getMin(); // return -2

```

1 class MinStack:
2
3     def __init__(self):
4         self.main_stack = []
5         self.min_stack = []
6
7     def push(self, val: int) -> None:
8         self.main_stack.append(val)
9         if not self.min_stack or val <= self.min_stack[-1]:
10             self.min_stack.append(val)
11
12     def pop(self) -> None:
13         if self.main_stack:
14             val = self.main_stack.pop()
15             if self.min_stack and val == self.min_stack[-1]:
16                 self.min_stack.pop()
17
18     def top(self) -> int:
19         if self.main_stack:
20             return self.main_stack[-1]
21
22     def getMin(self) -> int:
23         if self.min_stack:
24             return self.min_stack[-1]
25

```

Testcase Test Result

Accepted Runtime: 46 ms

Case 1

Input

["MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"]

[[], [-2], [0], [-3], [], [], [], []]

Code:

```

class MinStack:

    def __init__(self):

        self.main_stack = []

        self.min_stack = []

    def push(self, val: int) -> None:

        self.main_stack.append(val)

        if not self.min_stack or val <= self.min_stack[-1]:

            self.min_stack.append(val)

    def pop(self) -> None:

        if self.main_stack:

            val = self.main_stack.pop()

            if self.min_stack and val == self.min_stack[-1]:

```

```

        self.min_stack.pop()

def top(self) -> int:
    if self.main_stack:
        return self.main_stack[-1]

def getMin(self) -> int:
    if self.min_stack:
        return self.min_stack[-1]

```

Example:

Input:

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
```

```
[[],[-2],[0],[-3],[],[],[],[]]
```

```
# Output: [null,null,null,null,-3,null,0,-2]
```

## 4.6 Key Takeaways from the Resources

- Stacks follow the Last In, First Out principle, making them ideal for managing nested or recursive tasks.
- Linked list-based stacks allow dynamic resizing, whereas array-based stacks have fixed sizes.
- Stacks are efficient for insertion and deletion operations at one end but limit access to the top element only.
- Stacks are fundamental to algorithms involving expression evaluation, backtracking, and function call management.



## 5. Queues

### 5.1 Introduction to Queues

**Key Concept:** A queue is a linear data structure that follows the **FIFO (First In First Out)** principle, meaning the first element added is the first one to be removed. Queues are widely used in scenarios where items need to be processed in the order they were added.

**Structure:**

- **Front:** The end from which elements are removed.
- **Rear (or End):** The end where elements are added.
- **Enqueue:** The operation to add an element to the rear of the queue.
- **Dequeue:** The operation to remove an element from the front of the queue.

Queues can be implemented using arrays or linked lists and can also be circular.

### 5.2 Memory Allocation

Queues can be implemented using:

- **Arrays:** Allocate a fixed size of memory at initialization. Array-based queues have a fixed size, and you may need to manage overflow and underflow conditions.
- **Linked Lists:** Use dynamic memory allocation, where nodes are added as needed. This provides flexibility in the size of the queue but introduces extra overhead due to pointers.
- **Circular Queue:** A variation of the queue where the end of the queue wraps around to the beginning, making efficient use of space and avoiding overflow issues in fixed-size queues.

### 5.3 Applications, Benefits and Limitations of Stacks

**Applications:**

- Queues are used in operating systems for job scheduling and process management.
- Queues can be used in Breadth-First Search (BFS) which is essential in graph algorithms for exploring nodes level by level.
- Queues are utilized in buffering and caching systems, like printer queues and network data buffers.
- They are useful in scenarios where requests need to be processed in the order they are received.

**Benefits:**

- FIFO ensures fair processing of elements in the order they are added.
- It is easy to implement and understand with straightforward enqueue and dequeue operations.
- Linked list-based queues can dynamically grow or shrink, while circular queues optimize space usage.

**Limitations:**

- Static queues(array based) have a fixed size, which can lead to overflow if not managed properly.
- Linked list-based queues require additional memory for pointers, which can be inefficient.
- Circular queues require careful management of indices to avoid confusion and errors.
- Searching an element can take  $O(n)$  time.

### 5.4 Basic Operations

Basic operations performed on a queue are:

- **Enqueue Operation:** Add an element to the rear of the queue.
- **Dequeue Operation:** Remove an element from the front of the queue.
- **Peek Operation:** Retrieve the front element without removing it.
- **IsEmpty Operation:** Check if the queue is empty.
- **IsFull Operation:** Check if the queue is full (for array-based queues).

```
class Queue:
    def __init__(self, capacity):
```

```
self.queue = []
self.capacity = capacity

# Enqueue operation
def enqueue(self, value):
    if len(self.queue) < self.capacity:
        self.queue.append(value)
    else:
        print("Queue Overflow")

# Dequeue operation
def dequeue(self):
    if not self.is_empty():
        return self.queue.pop(0)
    else:
        print("Queue Underflow")
        return None

# Peek operation
def peek(self):
    if not self.is_empty():
        return self.queue[0]
    else:
        return None

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

# IsFull operation (for array-based queues)
def is_full(self):
    return len(self.queue) == self.capacity

# Example usage
queue = Queue(5)
queue.enqueue(1)
queue.enqueue(2)
print(queue.dequeue()) # Outputs: 1
print(queue.peek())   # Outputs: 2
```

## 5.5 Hands-on Coding Exercises

### 5.5.1 Problem 1: Implement Queue using Stacks

#### Problem Statement([Link](#)):

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

- `void push(int x)` Pushes element x to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns true if the queue is empty, false otherwise.

#### Notes:

- You must use **only** standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

#### Challenges Faced:

- It is a challenge to use only stacks to simulate queue behaviour, especially ensuring that the order of elements is maintained correctly for pop and peek operations. Here, using two stacks helps in reversing the order of elements, mimicking the FIFO behaviour of a queue.
- Next challenge is transferring elements between stacks when needed without compromising the queue's order. This is handled by transferring elements from `stack_in` to `stack_out` only when `stack_out` is empty, ensuring that the transfer operation is performed only when necessary.
- It is important to ensure that the `empty()` function correctly reflects the state of the queue, taking into account elements in both stacks. The empty method checks both `stack_in` and `stack_out` to determine if the queue is empty.
- Another challenge is to manage scenarios where multiple pop or peek operations are called consecutively. Proper handling of element transfers ensures that consecutive operations work as expected.

#### Approach:

- The solution involves using two stacks to simulate a queue. The two stacks are used to reverse the order of elements, allowing the operations of a queue (FIFO) to be achieved using stack operations (LIFO).
  - `stack_in`: Used to push new elements onto the queue.
  - `stack_out`: Used to pop or peek elements from the front of the queue.
- Push: Simply push the new element onto `stack_in`.
- Pop: If `stack_out` is empty, transfer all elements from `stack_in` to `stack_out` (reversing their order). Pop the top element from `stack_out`.
- Peek: If `stack_out` is empty, transfer all elements from `stack_in` to `stack_out` (reversing their order). Return the top element from `stack_out` without removing it.
- Empty: The queue is empty if both `stack_in` and `stack_out` are empty.
- The challenge is to simulate the FIFO behavior of a queue using the LIFO behavior of stacks. By using two stacks, the elements can be transferred and reversed to achieve the desired order.
- Each element is pushed and popped from `stack_in` and `stack_out` at most once, leading to amortized  $O(1)$  time complexity for push, pop, and peek operations.

#### Time Complexity:

- **push(x)**:  $O(1)$  - Pushing an element onto `stack_in` is a constant-time operation.
- **pop()**:  $O(1)$  amortized - Each element is pushed and popped from `stack_in` and `stack_out` at most once. Transfer operations are infrequent and do not affect the amortized complexity.
- **peek()**:  $O(1)$  amortized - Similar to pop, accessing the top element of `stack_out` is efficient.
- **empty()**:  $O(1)$  - Checking if both stacks are empty is a constant-time operation.

#### Solution:

### 232. Implement Queue using Stacks

Easy Topics Companies

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element `x` to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

**Notes:**

- You must use **only** standard operations of a stack, which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

**Example 1:**

**Input**  
["MyQueue", "push", "push", "peek", "pop", "empty"]  
[[], [1], [2], [], [], []]

**Output**  
[null, null, null, 1, 1, false]

**Explanation**  
MyQueue myQueue = new MyQueue();  
myQueue.push(1); // queue is: [1]  
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)  
myQueue.peek(); // return 1  
myQueue.pop(); // return 1, queue is [2]  
myQueue.empty(); // return false

```
1 class MyQueue:
2
3     def __init__(self):
4         self.stack_in = []
5         self.stack_out = []
6
7     def push(self, x: int) -> None:
8         self.stack_in.append(x)
9
10    def pop(self) -> int:
11        if not self.stack_out:
12            while self.stack_in:
13                self.stack_out.append(self.stack_in.pop())
14        return self.stack_out.pop()
15
16    def peek(self) -> int:
17        if not self.stack_out:
18            while self.stack_in:
19                self.stack_out.append(self.stack_in.pop())
20        return self.stack_out[-1]
21
22
23
24    def empty(self) -> bool:
25        return not self.stack_in and not self.stack_out
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Ln 32, Col 23

Testcase 1 Test Result

Accepted Runtime: 30 ms

Case 1

Input

["MyQueue", "push", "push", "peek", "pop", "empty"]

Code:

```
class MyQueue:

    def __init__(self):
        self.stack_in = []
        self.stack_out = []

    def push(self, x: int) -> None:
        self.stack_in.append(x)

    def pop(self) -> int:
        if not self.stack_out:
            while self.stack_in:
                self.stack_out.append(self.stack_in.pop())
        return self.stack_out.pop()

    def peek(self) -> int:
        if not self.stack_out:
            while self.stack_in:
                self.stack_out.append(self.stack_in.pop())
        return self.stack_out[-1]

    def empty(self) -> bool:
```

```
return not self.stack_in and not self.stack_out
```

Example:

Inout:

```
["MyQueue","push","push","peek","pop","empty"]
```

```
[[],[1],[2],[,],[],[]]
```

```
# Output: [null,null,null,1,1,false]
```

### 5.5.2 Problem 2: Moving Average from Data Stream

#### Problem Statement([Link](#)):

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Implement the MovingAverage class:

- MovingAverage(int size) Initializes the object with the size of the window size.
- double next(int val) Returns the moving average of the last size values of the stream.

#### Challenges Faced:

- First challenge is to make sure that each operation (next) runs in constant time, despite needing to maintain a sliding window of the last  $n$  elements. Using a queue and maintaining a running sum allowed for quick updates and avoided the need to recalculate the sum from scratch on every call.
- When the number of elements in the queue exceeds the window size, it is important to remove the oldest element while updating the sum. The solution dequeues the oldest element and adjusts the sum accordingly.
- It is a challenge to handle the edge cases such as when the queue has fewer elements than the window size (i.e., during the first few calls to next). The solution adjusts the average calculation to divide by the actual number of elements in the queue during the initial phases when the queue hasn't reached full capacity.

#### Approach:

- The goal is to maintain a sliding window of the last  $n$  values from the stream. Each time a new value is added, if the number of values exceeds  $n$ , the oldest value is removed from the window, and the average of the remaining values is returned.
- The approach uses a queue (FIFO structure) to store the most recent  $n$  values. When the size of the queue exceeds  $n$ , the oldest value is removed (the front of the queue).
- It then maintains a running sum of the values in the queue to quickly calculate the moving average. When a value is added to the queue, the sum is updated, and when a value is removed, the sum is decreased by the removed value.
- In the end, the moving average is returned by dividing the current sum by the number of elements in the queue.

#### Why Queue?

- A queue is a natural fit for the sliding window problem, as it allows for the efficient addition of new elements at the end and removal of the oldest elements from the front, maintaining the FIFO behaviour needed for the window.
- By keeping a running sum, the moving average is computed in constant time  $O(1)$  for each call to next, as we avoid recalculating the sum from scratch.

#### Time Complexity:

- **next(val):**  $O(1)$  - Each call to next involves adding a new element to the queue, possibly removing an old element, and updating the running sum, all of which are constant-time operations.

#### Solution:

### 346. Moving Average from Data Stream Premium

Easy Topics Companies

Solved

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Implement the `MovingAverage` class:

- `MovingAverage(int size)` Initializes the object with the size of the window `size`.
- `double next(int val)` Returns the moving average of the last `size` values of the stream.

**Example 1:**

**Input**  
["MovingAverage", "next", "next", "next", "next"]  
[[3], [1], [10], [3], [5]]

**Output**  
[null, 1.0, 5.5, 4.66667, 6.0]

**Explanation**  
MovingAverage movingAverage = new MovingAverage(3);  
movingAverage.next(1); // return 1.0 = 1 / 1  
movingAverage.next(10); // return 5.5 = (1 + 10) / 2  
movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3  
movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3

**Constraints:**

- $1 \leq \text{size} \leq 1000$
- $-10^5 \leq \text{val} \leq 10^5$
- At most  $10^5$  calls will be made to `next`.

```

1 class MovingAverage:
2
3     def __init__(self, size: int):
4         self.size = size
5         self.queue = deque() # to store the sliding window values
6         self.sum = 0 # to keep track of the sum of elements in the window
7
8     def next(self, val: int) -> float:
9         # Add the new value to the queue and update the sum
10        self.queue.append(val)
11        self.sum += val
12
13        # If the queue exceeds the window size, remove the oldest element
14        if len(self.queue) > self.size:
15            oldest = self.queue.popleft()
16            self.sum -= oldest
17
18        # Return the moving average
19        return self.sum / len(self.queue)
20
21

```

Ln 20, Col 42

Testcase Test Result

Accepted Runtime: 30 ms

Case 1

Input

["MovingAverage", "next", "next", "next", "next"]

[[3], [1], [10], [3], [5]]

Code:

```

class MovingAverage:

    def __init__(self, size: int):

        self.size = size

        self.queue = deque() # to store the sliding window values

        self.sum = 0 # to keep track of the sum of elements in the window

    def next(self, val: int) -> float:

        # Add the new value to the queue and update the sum

        self.queue.append(val)

        self.sum += val

        # If the queue exceeds the window size, remove the oldest element

        if len(self.queue) > self.size:

            oldest = self.queue.popleft()

            self.sum -= oldest

        # Return the moving average

        return self.sum / len(self.queue)

```

Example:

Input:

```

["MovingAverage", "next", "next", "next", "next"]
[[3], [1], [10], [3], [5]]

```

```
# Output: [null,1.00000,5.50000,4.66667,6.00000]
```

## 5.6 Key Takeaways from the Resources

- Queues adhere to the First In, First Out principle, making them suitable for scenarios where processing order is crucial.
- Queues can be implemented using arrays, linked lists, or circular structures, each with different trade-offs in terms of efficiency and complexity.
- Queues are essential in task scheduling, breadth-first search, and buffering applications, demonstrating their practical utility in various domains.
- Array-based queues have a fixed size which can lead to overflow, while linked list and circular queues offer more flexibility and efficient space utilization.

## 6. Hash Tables

### 6.1 Introduction to Hash Tables

**Key Concept:** A hash table is a data structure that implements an associative array, a structure that can map keys to values. It provides efficient insertion, deletion, and lookup operations based on a hash function. Hash tables are widely used for implementing dictionaries, caches, and sets.

**Structure:**

- **Hash Function:** A function that converts a key into an index in an array. The goal is to distribute keys uniformly across the table to minimize collisions.
- **Hash Code:** The hash function crunches the data and give a unique hash code. This hash code is typically integer value that can be used an index.
- **Buckets (or Slots):** Array elements that store values. Each bucket can hold multiple entries if collisions occur.
- **Collision Handling:** Techniques used to manage cases where multiple keys hash to the same index, such as chaining (linked lists) or open addressing (probing).

### 6.2 Memory Allocation

Hash tables use contiguous memory allocation for the array of buckets. Memory is allocated based on the size of the table and the number of buckets.

- **Dynamic Resizing:** Hash tables often resize themselves dynamically to maintain performance. When the load factor (ratio of the number of elements to the number of buckets) exceeds a threshold, the table is resized, and elements are rehashed.

### 6.3 Applications, Benefits and Limitations of Stacks

**Applications:**

- Hash tables are used to quickly locate data without scanning the entire database.
- Hash tables are used in caching systems to store and retrieve data efficiently.
- Hashing is used in database indexing.
- Hashing functions are used to store passwords on a website
- Hash functions are implemented in sets to manage unique elements and perform operations like union, intersection, and difference.

**Benefits:**

- Average time complexity for insertion, deletion, and lookup operations is  $O(1)$  due to the direct indexing provided by the hash function.
- Can handle dynamic data sizes efficiently through resizing and rehashing.
- Suitable for various applications requiring fast access to data.

**Limitations:**

- Hash collisions can occur when multiple keys hash to the same index. Proper collision handling strategies are required to maintain performance.
- Hash tables can use more memory than other data structures due to the overhead of maintaining buckets and handling collisions.
- The efficiency of a hash table heavily relies on the quality of the hash function. Poor hash functions can lead to clustering and degraded performance.

### 6.4 Basic Operations

Basic operations performed on a hash table are:

- **Insertion:** Add a key-value pair to the hash table.
- **Deletion:** Remove a key-value pair from the hash table.
- **Search:** Find the value associated with a given key.
- **Rehashing:** Resize and rehash the table when the load factor exceeds a threshold.



```
class HashTable:

    def __init__(self, size):

        self.size = size

        self.table = [[] for _ in range(size)]

    def hash_function(self, key):

        return hash(key) % self.size

    # Insert operation
    def insert(self, key, value):

        index = self.hash_function(key)

        for i, kv in enumerate(self.table[index]):

            if kv[0] == key:

                self.table[index][i] = (key, value)

                return

        self.table[index].append((key, value))

    # Search operation
    def search(self, key):

        index = self.hash_function(key)

        for kv in self.table[index]:

            if kv[0] == key:

                return kv[1]

        return None

    # Delete operation
    def delete(self, key):

        index = self.hash_function(key)

        for i, kv in enumerate(self.table[index]):

            if kv[0] == key:

                del self.table[index][i]

                return

# Example usage
hash_table = HashTable(10)
hash_table.insert("apple", 1)
hash_table.insert("banana", 2)
print(hash_table.search("apple")) # Outputs: 1
hash_table.delete("apple")
print(hash_table.search("apple")) # Outputs: None
```

## 6.5 Hands-on Coding Exercises

### 6.5.1 Problem 1: Single Number

#### Problem Statement([Link](#)):

Given a non-empty array of integers `nums`, every element appears *twice* except for one. Find that single one. You must implement a solution with a linear runtime complexity and use only constant extra space.

#### Challenges Faced:

- A challenge was identifying the most appropriate data structure that can store frequencies efficiently while allowing fast lookups. A hash table is ideal because it provides  $O(1)$  average time complexity for insertions and lookups.
- It is a challenge to handle cases such as an array with only one element, or ensuring that duplicate elements are correctly counted in the hash table.

#### Approach:

- To solve the problem using a hash table, we can take advantage of the ability to store the frequency (count) of each element.
- The array is traversed and a hash table is used to count how many times each element appears.
- In a second pass, it is checked which element appears exactly once, and that element is returned.

#### Time Complexity:

- $O(n)$  — We traverse the array twice, which makes the solution linear with respect to the size of the array.

#### Solution:

The screenshot displays a coding interface for the '136. Single Number' problem. On the left, the problem description states: 'Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single one. You must implement a solution with a linear runtime complexity and use only constant extra space.' It includes three examples: Example 1 (Input: [2,2,1], Output: 1), Example 2 (Input: [4,1,2,1,2], Output: 4), and Example 3 (Input: [1], Output: 1). Constraints are listed as  $1 \leq \text{nums.length} \leq 3 \times 10^4$ ,  $-3 \times 10^4 \leq \text{nums}[i] \leq 3 \times 10^4$ , and 'Each element in the array appears twice except for one element which appears only once.' On the right, a Python3 code editor shows the solution: 

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        freq = {}
        for num in nums:
            if num in freq:
                freq[num] += 1
            else:
                freq[num] = 1
        for num in nums:
            if freq[num] == 1:
                return num
```

 Below the code, the 'Test Result' section shows 'Accepted' with a runtime of 44 ms and three test cases passed.

#### Code:

```
def singleNumber(nums):
    freq = {}

    for num in nums:
        if num in freq:
            freq[num] += 1
        else:
            freq[num] = 1

    for num in nums:
```

```
if freq[num] == 1:  
    return num
```

Example:

```
nums = [2,2,1]  
singleNumber(nums)  
  
# Output: 1
```

### 6.5.2 Problem 2: Isomorphic Strings

#### Problem Statement([Link](#)):

Given two strings  $s$  and  $t$ , *determine if they are isomorphic*.

Two strings  $s$  and  $t$  are isomorphic if the characters in  $s$  can be replaced to get  $t$ .

All occurrences of a character must be replaced with another character while preserving the order of characters.

No two characters may map to the same character, but a character may map to itself.

#### Challenges Faced:

- One challenge was figuring out how to ensure that each character in  $s$  maps to a consistent character in  $t$ , while preventing multiple characters in  $s$  from mapping to the same character in  $t$ .
- Another challenge was to handle cases where the lengths of the strings are unequal, or where multiple characters in  $s$  need to map to the same character in  $t$ .

#### Approach:

- Hash table(dictionary) is used to determine if two strings are isomorphic, to map characters from string  $s$  to string  $t$  and check for consistency.
- Here two hash tables are used(one for  $s \rightarrow t$  mapping and another for  $t \rightarrow s$  mapping).
- Both strings are iterated simultaneously, checking if the characters in  $s$  can map to characters in  $t$  and vice versa.
- If at any point, the mappings are inconsistent (i.e., a character in  $s$  maps to different characters in  $t$  or vice versa), the strings are not isomorphic.
- If the strings have different lengths, the program returns False immediately because they cannot be isomorphic.
- Hash tables allows to quickly check and store mappings from characters in  $s$  to characters in  $t$  and the reverse mapping.
- Two hash tables are used to ensure that not only does every character in  $s$  map consistently to a character in  $t$ , but also that no two characters from  $s$  map to the same character in  $t$ .

#### Time Complexity:

- $O(n)$  — We traverse the array twice, which makes the solution linear with respect to the size of the array.

#### Solution:

## 205. Isomorphic Strings

Easy Topics Companies

Given two strings `s` and `t`, determine if they are isomorphic.

Two strings `s` and `t` are isomorphic if the characters in `s` can be replaced to get `t`.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

**Example 1:**

Input: `s = "egg", t = "add"`

Output: `true`

Explanation:

The strings `s` and `t` can be made identical by:

- Mapping `'e'` to `'a'`.
- Mapping `'g'` to `'d'`.

**Example 2:**

Input: `s = "foo", t = "bar"`

Output: `false`

Explanation:

The strings `s` and `t` can not be made identical as `'o'` needs to be mapped to both `'a'` and `'r'`.

**Example 3:**

Input: `s = "paper", t = "title"`

Output: `true`

```
1 class Solution:
2     def isIsomorphic(self, s: str, t: str) -> bool:
3         if len(s) != len(t):
4             return False
5
6         map_s_to_t = {}
7         map_t_to_s = {}
8
9         for char_s, char_t in zip(s, t):
10             if char_s in map_s_to_t:
11                 if map_s_to_t[char_s] != char_t:
12                     return False
13             else:
14                 map_s_to_t[char_s] = char_t
15
16             if char_t in map_t_to_s:
17                 if map_t_to_s[char_t] != char_s:
18                     return False
19             else:
20                 map_t_to_s[char_t] = char_s
21
22         return True
```

Testcase Test Result

Accepted Runtime: 44 ms

Case 1 Case 2 Case 3

Input

`s =`

`"foo"`

`t =`

`"pap"`

Code:

```
def isIsomorphic(s, t):
    if len(s) != len(t):
        return False

    map_s_to_t = {}
    map_t_to_s = {}

    for char_s, char_t in zip(s, t):
        if char_s in map_s_to_t:
            if map_s_to_t[char_s] != char_t:
                return False
        else:
            map_s_to_t[char_s] = char_t

        if char_t in map_t_to_s:
            if map_t_to_s[char_t] != char_s:
                return False
        else:
            map_t_to_s[char_t] = char_s

    return True
```

Example 1:

```
s = "egg"
```

```
t = "add"
isIsomorphic(s, t)

# Output: True
```

Example 2:

```
s = "foo"
t = "bar"
isIsomorphic(s, t)

# Output: False
```

## 6.6 Key Takeaways from the Resources

- Hash tables use a hash function to map keys to indices, providing efficient average-time complexity for operations.
- Handling collisions is crucial for maintaining the performance of hash tables, with common strategies including chaining and open addressing.
- Resizing and rehashing are used to manage the load factor and ensure efficient operation as the number of elements grows.
- Hash tables are versatile and widely used in various applications such as caching, indexing, and set operations due to their efficiency and flexibility.

## 7. Algorithms

### 7.1 Searching Algorithms

#### 7.1.1 Introduction to Searching

Searching is a fundamental operation in computer science and data structures, where the goal is to find a specific element or value within a data structure (e.g., arrays, linked lists, or trees). Searching algorithms are essential for applications involving large datasets, where locating an item efficiently is critical for performance.

##### Importance in DSA:

- Searching is used in numerous real-world applications, such as databases, file systems, networking, and more.
- Fast searching algorithms improve the efficiency of operations like querying, sorting, and managing data.
- A well-chosen search algorithm can significantly reduce time complexity.

##### Characteristics of Searching Algorithms

- **Target Element:** In searching, there is always a specific target element or item that has to be found within the data collection. This target could be a value, a record, a key, or any other data entity of interest.
- **Search Space:** The search space refers to the entire collection of data within which the target element is searched. Depending on the data structure used, the search space may vary in size and organization.
- **Time and Space Complexity:** Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.
- **Deterministic Vs Non-deterministic:** Some searching algorithms, like binary search are deterministic, meaning they follow a clear and systematic approach. Others, such as linear search, are non-deterministic, as they may need to examine the entire search space in the worst case.

#### 7.1.2 Applications, Benefits and Limitations of Searching

##### Applications:

- Searching is used in database management systems for retrieving specific data records based on user queries
- File systems for locating and retrieving data.
- Searching is crucial in e-commerce platforms for users to find products quickly based on their preferences, specifications, or keywords.
- Search engines like Google, Bing, and Yahoo to retrieve relevant information from vast amounts of data on the web.
- In networking, searching algorithms are used for routing packets efficiently through networks, finding optimal paths.

##### Benefits:

- Efficient searching can speed up data retrieval, saving time in applications where large datasets are involved.
- Specialized algorithms (e.g., binary search) offer logarithmic search time, making them suitable for sorted data.

##### Limitations:

- Some algorithms (e.g., linear search) have poor performance on large datasets due to  $O(n)$  time complexity.
- Many searching algorithms (e.g., binary search) require pre-sorted data to function optimally.

#### 7.1.3 Searching Algorithms

Some of the common searching algorithms are as follows:

##### 1. Linear Search

This algorithm scans each element in the array sequentially until the target element is found or the end of the array is reached. The time complexity of this algorithm is  $O(n)$  and space complexity is  $O(1)$ . It is suitable for unsorted array or if the dataset is very small.

```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

# Example
arr = [2, 3, 4, 10, 40]
x = 10
print("Element found at index", linear_search(arr, x))

# Output: Element found at index 3
```

## 2. Binary Search

This algorithm works by dividing the array into two halves, repeatedly eliminating the half where the target element cannot reside. Binary Search requires that the array is already sorted. The time complexity of this algorithm is  $O(\log n)$  and space complexity is  $O(1)$ .

```
def binary_search(arr, l, r, x):
    while l <= r:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            l = mid + 1
        else:
            r = mid - 1
    return -1

# Example
arr = [2, 3, 4, 10, 40]
x = 10
print("Element found at index", binary_search(arr, 0, len(arr)-1, x))

# Output: Element found at index 3
```

## 3. Jump Search

Jump search skips ahead by fixed steps (jumping) and then performs a linear search within the skipped block when the element is less than the jump index. The time complexity of this algorithm is  $O(\sqrt{n})$  and space complexity is  $O(1)$ . The array has to be sorted for this searching algorithm.

```
import math
```

```
def jump_search(arr, x):
    n = len(arr)
    step = math.sqrt(n)
    prev = 0
    while arr[int(min(step, n)-1)] < x:
        prev = step
        step += math.sqrt(n)
        if prev >= n:
            return -1
    for i in range(int(prev), int(min(step, n))):
        if arr[i] == x:
            return i
    return -1

# Example
arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
x = 5
print("Element found at index", jump_search(arr, x))

# Output: Element found at index 5
```

#### 4. Interpolation Search

This algorithm is an advanced search algorithm that uses the position of the target element based on its value, working best for uniformly distributed data. The time complexity is  $O(\log \log n)$  (best case) and  $O(n)$  (worst case).

```
def interpolation_search(arr, n, x):
    lo = 0
    hi = n - 1
    while lo <= hi and x >= arr[lo] and x <= arr[hi]:
        if lo == hi:
            if arr[lo] == x:
                return lo
            return -1
        pos = lo + int(((float(hi - lo) / (arr[hi] - arr[lo])) * (x - arr[lo])))
        if arr[pos] == x:
            return pos
        if arr[pos] < x:
            lo = pos + 1
        else:
            hi = pos - 1
```



```

    return -1

# Example
arr = [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47]
x = 18
print("Element found at index", interpolation_search(arr, len(arr), x))

# Output: Element found at index 4

```

## 5. Exponential Search

Exponential search combines binary search with an exponential increase in steps to locate the target element efficiently. The time complexity for this algorithm is  $O(\log n)$  and space complexity is  $O(1)$ .

```

def exponential_search(arr, n, x):
    if arr[0] == x:
        return 0
    i = 1
    while i < n and arr[i] <= x:
        i = i * 2
    return binary_search(arr, i//2, min(i, n-1), x)

# Example
arr = [10, 20, 40, 45, 55, 60, 70, 80]
x = 55
print("Element found at index", exponential_search(arr, len(arr), x))

# Output: Element found at index 4

```

## 6. Ternary Search

Ternary Search is a divide-and-conquer algorithm, similar to binary search, but instead of dividing the array into two parts, it divides it into three. It recursively checks the two midpoints of the array and eliminates one-third of the array in each step. The algorithm assumes the array is sorted.

- It calculates 2 mid points:
  - $\text{mid1} = \text{left} + (\text{right} - \text{left}) // 3$
  - $\text{mid2} = \text{right} - (\text{right} - \text{left}) // 3$
- Compare the target value with the elements at mid1 and mid2.
- Depending on where the target falls, eliminate one-third of the search space and recursively search in the remaining two-thirds.

Time Complexity of this algorithm is  $O(\log_3 n)$  in worst case, since the array is divided into three parts instead of two (as in binary search), it is slightly slower than binary search but still logarithmic. Ternary search can be useful in specific optimization problems or cases where a triple partitioning makes sense, but in most general-purpose searching tasks, binary search is preferred because of its better practical performance.

```

def ternary_search(arr, left, right, key):
    if right >= left:

```

```

mid1 = left + (right - left) // 3
mid2 = right - (right - left) // 3

if arr[mid1] == key:
    return mid1
if arr[mid2] == key:
    return mid2

if key < arr[mid1]:
    return ternary_search(arr, left, mid1 - 1, key)
elif key > arr[mid2]:
    return ternary_search(arr, mid2 + 1, right, key)
else:
    return ternary_search(arr, mid1 + 1, mid2 - 1, key)

return -1

# Example
arr = [1, 3, 5, 7, 9, 11, 13, 15]
key = 7
result = ternary_search(arr, 0, len(arr)-1, key)
print("Element found at index", result) if result != -1 else print("Element not found")

# Output: Element found at index 3

```

## 7.2 Sorting Algorithms

### 7.2.1 Introduction to Sorting

**Sorting** is the process of arranging data in a specific order, usually in ascending or descending order. It is a fundamental concept in computer science, used for organizing and optimizing data structures to improve performance during data manipulation, searching, and retrieval.

Sorting allows for efficient data analysis, reducing the complexity of certain tasks, such as binary search and merging datasets.

#### Characteristics of Sorting

1. **Stability:** A sorting algorithm is stable if it preserves the relative order of elements with equal keys.
2. **In-Place:** A sorting algorithm is in-place if it requires only a constant amount of extra space besides the input data.
3. **Time Complexity:** Refers to how the algorithm's run-time increases with the size of the input. Common measures include best, average, and worst-case scenarios.
4. **Adaptive:** An adaptive sorting algorithm can take advantage of an already partially sorted input, reducing the number of operations required to sort the data completely.

### 7.2.2 Applications, Benefits and Limitations of Sorting

#### Applications:

- Efficient sorting allows faster searching algorithms like binary search to be applied.
- Sorting helps in data analysis by arranging data for better readability, especially for statistical and analytical purposes.
- Many algorithms, like divide-and-conquer and dynamic programming approaches, rely on sorted input for efficient execution.
- Sorting is used in database indexing, which speeds up query processing.

#### Benefits:

- Sorted data can be searched more efficiently (e.g., using binary search).
- Sorted data makes data presentation more readable and meaningful.
- Many algorithms (e.g., merge, binary search, and some graph algorithms) require data to be sorted for optimal performance.

#### Limitations:

- Sorting can be computationally expensive for large datasets, especially for inefficient algorithms (e.g.,  $O(n^2)$  algorithms like Bubble Sort).
- Non-in-place algorithms may consume additional memory, which could be a limitation in resource-constrained environments.
- Some sorting algorithms do not preserve the original order of equal elements.

### 7.2.3 Sorting Algorithms

Sorting algorithms are classified as:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)

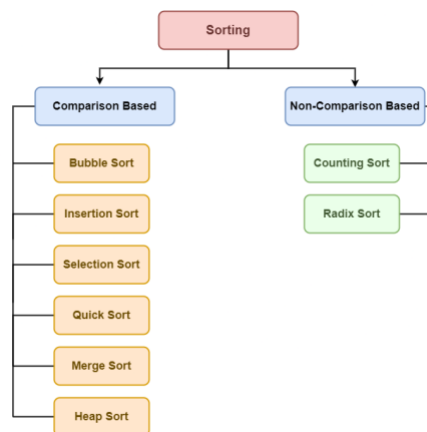


Figure 5: Source: [geeksforgeeks.com](https://www.geeksforgeeks.com)

### Comparison-based

#### 1. Bubble Sort:

Bubble sort algorithm is a stable in-place algorithm which compares adjacent elements and swaps them if they are in the wrong order. This process repeats until the array is sorted. The time complexity for this algorithm is  $O(n^2)$  in worst case,  $O(n)$  in best case (if already sorted). It is suitable for small datasets where simplicity is more important than performance.

```

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

# Example
arr = [64, 34, 25, 12, 22, 11, 90]
print("Bubble Sort:", bubble_sort(arr))    # Output: Bubble Sort: [11, 12, 22, 25, 34, 64, 90]

```

## 2. Selection Sort:

Selection sort algorithm is an in-place algorithm which finds the minimum element from the unsorted part of the array and swaps it with the first unsorted element. This process continues until the entire array is sorted. The time complexity for this algorithm is  $O(n^2)$ . It is suitable for sorting small arrays.

```

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

# Example
arr = [64, 25, 12, 22, 11]
print("Selection Sort:", selection_sort(arr))    # Output: Selection Sort: [11, 12, 22, 25, 64]

```

## 3. Insertion Sort:

Insertion sort algorithm is an in-place algorithm which builds the sorted array one element at a time by comparing and inserting elements into their correct positions. The time complexity for this algorithm is  $O(n^2)$  in worst case,  $O(n)$  for nearly sorted data. It is efficient for small datasets or nearly sorted datasets.

```

def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        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

```

```
# Example
```

```
arr = [12, 11, 13, 5, 6]
```

```
print("Insertion Sort:", insertion_sort(arr))    # Output: Insertion Sort: [5, 6, 11, 12, 13]
```

#### 4. Merge Sort:

Merge sort algorithm is a divide-and-conquer algorithm that splits the array into halves, recursively sorts them, and then merges the sorted halves. Merge Sort is not an in-place algorithm but it is stable. The time complexity for this algorithm is  $O(n \log n)$ . It is suitable for large datasets due to its  $O(n \log n)$  time complexity.

```
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
```

# Example

```
arr = [12, 11, 13, 5, 6, 7]
```

```
print("Merge Sort:", merge_sort(arr))      # Output: Merge Sort: [5, 6, 7, 11, 12, 13]
```

## 5. Quick Sort:

Quick sort algorithm is also a divide-and-conquer algorithm that selects a "pivot" element and partitions the array into two sub-arrays based on the pivot. It recursively sorts the sub-arrays. The time complexity for this algorithm is  $O(n \log n)$  on average,  $O(n^2)$  in worst case. It is an in-place algorithm that is generally the fastest sorting algorithm for average cases and widely used in practice.

```
def partition(arr, low, high):
```

```
    pivot = arr[high]
```

```
    i = low - 1
```

```
    for j in range(low, high):
```

```
        if arr[j] <= pivot:
```

```
            i += 1
```

```
            arr[i], arr[j] = arr[j], arr[i]
```

```
    arr[i+1], arr[high] = arr[high], arr[i+1]
```

```
    return i+1
```

```
def quick_sort(arr, low, high):
```

```
    if low < high:
```

```
        pi = partition(arr, low, high)
```

```
        quick_sort(arr, low, pi-1)
```

```
        quick_sort(arr, pi+1, high)
```

```
    return arr
```

# Example

```
arr = [10, 7, 8, 9, 1, 5]
```

```
print("Quick Sort:", quick_sort(arr, 0, len(arr)-1))      # Output: Quick Sort: [1, 5, 7, 8, 9, 10]
```

## 6. Heap Sort:

Heap sort algorithm converts the array into a heap structure and then repeatedly extracts the maximum element from the heap, placing it in the correct position. The time complexity for this algorithm is  $O(n \log n)$ . It is an in-place algorithm which is suitable for large datasets when in-place sorting is needed and stability is not a concern.

```
def heapify(arr, n, i):
```

```
    largest = i
```

```
    left = 2 * i + 1
```

```
    right = 2 * i + 2
```

```

if left < n and arr[left] > arr[largest]:
    largest = left
if right < n and arr[right] > arr[largest]:
    largest = right
if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]
    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[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

# Example
arr = [12, 11, 13, 5, 6, 7]
print("Heap Sort:", heap_sort(arr))      # Output: Heap Sort: [5, 6, 7, 11, 12, 13]

```

## Non-Comparison Based

### 1. Counting Sort:

Counting sort algorithm is a non-comparison-based algorithm that counts occurrences of each unique element, then calculates positions and places elements in sorted order. The time complexity for this algorithm is  $O(n + k)$ , where  $k$  is the maximum possible value in the array. It is efficient for small integer arrays with a limited range of values.

```

def counting_sort(arr):
    max_val = max(arr)
    count = [0] * (max_val + 1)
    output = [0] * len(arr)

    for i in arr:
        count[i] += 1

    for i in range(1, len(count)):
        count[i] += count[i-1]

```

```

for i in range(len(arr)-1, -1, -1):
    output[count[arr[i]] - 1] = arr[i]
    count[arr[i]] -= 1

return output

# Example
arr = [1, 4, 1, 2, 7, 5, 2]
print("Counting Sort:", counting_sort(arr))    # Output: Counting Sort: [1, 1, 2, 2, 4, 5, 7]

```

## 2. Radix Sort:

Radix sort algorithm is a stable algorithm which sorts elements digit by digit, starting from the least significant digit. The time complexity for this algorithm is  $O(nk)$ , where  $k$  is the number of digits in the largest number. It is used for sorting large numbers or strings, especially when the size of the numbers or range of elements is large.

```

def counting_sort_exp(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in arr:
        index = i // exp
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    for i in range(n - 1, -1, -1):
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1

    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:

```



```

counting_sort_exp(arr, exp)

exp *= 10

return arr

# Example
arr = [170, 45, 75, 90, 802, 24, 2, 66]
print("Radix Sort:", radix_sort(arr))      # Output: Radix Sort: [2, 24, 45, 66, 75, 90, 170, 802]

```

## 7.3 Dijkstra Algorithm and Bellman-Ford Algorithm

Dijkstra's and Bellman-Ford algorithms are two fundamental approaches used to solve the shortest path problem in a weighted graph. While both help in determining the shortest path from a source node to all other nodes, they differ in handling certain types of graphs, particularly when negative edge weights are involved.

### 7.3.1 Dijkstra Algorithm

Dijkstra's algorithm is used to find the shortest path from a starting node (source) to all other nodes in a weighted graph where all edge weights are non-negative. Dijkstra's algorithm employs a *greedy* approach to find the shortest path between nodes in a graph. It starts with a source node, repeatedly picks the nearest unvisited node, and updates the shortest distances to other nodes.

#### Structure:

- A graph is represented using an adjacency list. The algorithm maintains a distance list where each entry stores the shortest distance from the source to that node. It uses a priority queue (min-heap) to choose the next nearest node at each step.

#### Approach:

1. First distances are initialized from the source node to all other nodes as infinity (`float('inf')`), and distance to the source itself as 0.
2. Then the source node is inserted into the priority queue.
3. The node with the smallest distance is extracted repeatedly and the distances to its neighbouring nodes are updated if a shorter path is found.
4. The step continues until all nodes are processed.

#### Time Complexity:

- With a min-heap, the time complexity is  $O((V + E) \log V)$  where  $V$  is the number of vertices and  $E$  is the number of edges.

Code:

```

import heapq

def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)

```

```

if current_distance > distances[current_node]:
    continue

for neighbor, weight in graph[current_node].items():
    distance = current_distance + weight

    if distance < distances[neighbor]:
        distances[neighbor] = distance
        heapq.heappush(pq, (distance, neighbor))

return distances

```

Example:

```

graph = {
    'A': {'B': 2, 'D': 1},
    'B': {'A': 2, 'C': 4, 'D': 3, 'E': 7},
    'C': {'B': 4, 'E': 6},
    'D': {'A': 1, 'B': 3, 'E': 5},
    'E': {'B': 7, 'C': 6, 'D': 5}
}

print(dijkstra(graph, 'A'))

# Output: {'A': 0, 'B': 2, 'C': 6, 'D': 1, 'E': 6}

```

### 7.3.2 Bellman-Ford Algorithm

Bellman-Ford is a dynamic programming algorithm that computes the shortest path from a single source to all other nodes. Unlike Dijkstra's, Bellman-Ford can handle graphs with negative edge weights and can detect negative-weight cycles.

#### Structure:

- Similar to Dijkstra's, Bellman-Ford represents the graph using an adjacency list, but it does not use a priority queue. Instead, it performs multiple *relaxations* of all edges, gradually finding the shortest path.

#### Approach:

1. First distances are initialized from the source to all other vertices as infinity, and distance to the source itself as 0.
2. Each edge is relaxed ( $V - 1$ ) times, where  $V$  is the number of vertices.
3. After ( $V - 1$ ) relaxations, the algorithm checks for negative-weight cycles. If a cycle is detected, the algorithm reports it.

#### Time Complexity:

- Bellman-Ford has a time complexity of  $O(V * E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges.

Code:

```
def bellman_ford(graph, V, source):
    dist = [float('inf')] * V
    dist[source] = 0

    for i in range(V - 1):
        for u, v, w in graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    for u, v, w in graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            return "Graph contains negative weight cycle"

    return dist
```

Example:

```
edges = [
    (0, 1, -1), (0, 3, 2), (1, 2, 4), (1, 3, -3),
    (3, 2, 6), (3, 4, 5), (1, 4, 3), (4, 2, 6)
]

print(bellman_ford(edges, 5, 0))

# Output: [0, -1, 2, -4, 1]
```

### 7.3.3 Dijkstra Algorithm Vs Bellman-Ford Algorithm

- Dijkstra's Algorithm utilizes a greedy approach to find the shortest path while Bellman-Ford Algorithm employs dynamic programming to calculate the shortest path.
- Dijkstra's Algorithm works only with graphs that have non-negative edge weights while Bellman-Ford Algorithm can handle graphs with negative edge weights.
- Dijkstra's Algorithm typically has a time complexity of  $O((V + E) \log V)$  when implemented with a priority queue (where  $V$  is vertices and  $E$  is edges) while Bellman-Ford Algorithm has a time complexity of  $O(V * E)$ , making it slower compared to Dijkstra's algorithm.
- Dijkstra's Algorithm cannot detect negative weight cycles in a graph but Bellman-Ford Algorithm can detect negative weight cycles and is often used in such scenarios.
- Dijkstra's Algorithm is more efficient for graphs with positive weights and fewer edges while Bellman-Ford Algorithm is slower due to its ability to handle negative weights and is preferred when negative weights are present in the graph.
- Both algorithms are designed to find the shortest path, but Bellman-Ford provides a more general solution at the cost of higher time complexity.

### 7.3.4 Applications of Dijkstra Algorithm and Bellman-Ford Algorithm

#### Applications

- **Dijkstra's Algorithm:**
  - GPS navigation systems.
  - Network routing protocols like OSPF (Open Shortest Path First).
  - Efficient pathfinding in maps or games.
- **Bellman-Ford Algorithm:**
  - Can be used in currency arbitrage detection due to its ability to detect negative cycles.
  - Suitable for networks with both positive and negative link costs.

### 7.3.5 Key Takeaways from references

- Dijkstra's algorithm is faster but limited to graphs without negative weights.
- Bellman-Ford is more versatile, and able to handle graphs with negative weights and cycles, though at the cost of increased time complexity.
- Both algorithms are critical for solving shortest path problems in graph theory, each with its distinct use cases depending on the graph's characteristics.

## 7.4 Graph Traversals(BFS and DFS)

Graph traversal refers to visiting all nodes (vertices) in a graph in a systematic manner. Two primary graph traversal methods are Breadth-First Search (BFS) and Depth-First Search (DFS), which explore nodes in different ways.

### Key Concepts

- **BFS:** Explores the graph level by level, starting from a source node and moving outward to adjacent nodes.
- **DFS:** Explores a graph by going as deep as possible along each branch before backtracking.

### 7.4.1 Breadth-First Search(BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores nodes level by level. It begins at a source node and explores all its immediate neighbors before moving on to the neighbors' neighbors. BFS is often used to find the shortest path in unweighted graphs.

### Characteristics

- The BFS algorithm uses a queue to store nodes for future exploration.
- It visits each node once, ensuring efficient exploration.
- Time complexity:  $O(V+E)$ , where  $V$  is vertices and  $E$  is edges.
- Space complexity:  $O(V)$  for storing the queue.

### Applications

- In an unweighted graph, BFS can be used to find the shortest path between two nodes.
- BFS are useful in detecting cycles in a graph
- In Peer-to-Peer Networks like BitTorrent Breadth First Search is used to find all neighbor nodes.
- Crawlers in Search Engines: Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same.
- In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
- Breadth First Search is used to find all neighboring locations in GPS Navigation systems
- In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

### Benefits and Limitations

- Guarantees the shortest path in unweighted graphs.
- If there is more than one solution then BFS can find the minimal one that requires less number of steps.
- Low storage requirement – linear with depth.
- Easily programmable.

```
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, end=" ")
            visited.add(node)
            queue.extend(graph[node] - visited)

# Example graph and BFS call
graph = {0: {1, 2}, 1: {2}, 2: {0, 3}, 3: {3}}
bfs(graph, 2)

# Output: 2 0 3 1
```

### 7.4.2 Depth-First Search(DFS)

Depth-First Search (DFS) is a graph traversal method that explores as deep as possible along each branch before backtracking. It uses either recursion or a stack data structure to maintain the exploration order.

#### Characteristics

- DFS uses a stack (or recursion) for traversal.
- It explores the depth of each node before backtracking.
- Time complexity:  $O(V+E)$  similar to BFS.
- Space complexity:  $O(V)$  due to the recursion stack or explicit stack.

#### Applications

- A graph has a cycle if and only if we see a back edge during DFS. So DFS could be used for the graph and back edges can be checked.
- DFS algorithm can be used to find a path between two given vertices  $u$  and  $z$ .
- It can be used to solve puzzles like mazes.
- It can be used to find connected components in a graph. A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- Depth-first search can be used in the implementation of web crawlers to explore the links on a website.
- Depth-first search can be used in backtracking algorithms.

#### Benefits

- Depth-First Search (DFS) is particularly useful for exploring deeper branches of a graph, focusing on fewer nodes at the same level compared to other strategies.
- DFS has a memory requirement that scales linearly with the size of the search graph. Unlike Breadth-First Search (BFS), which consumes more memory due to storing a queue of all nodes at the current level, DFS only needs to maintain a stack of nodes along the current search path.
- The time complexity of a depth-first Search to depth  $d$  and branching factor  $b$  (the number of children at each node, the outdegree) is  $O(b^d)$  since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.

- Because DFS only keeps track of nodes on the current path from the root to the current node, it uses less memory. In some cases, DFS may discover a solution without extensively exploring the entire search space.

### Limitations

- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
- May not find the shortest path in unweighted graphs.
- Can result in deep recursion and higher memory usage.

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()

    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)

# Example graph and DFS call
graph = {0: {1, 2}, 1: {2}, 2: {0, 3}, 3: {3}}
dfs(graph, 2)

# Output: 2 0 1 3
```

## 7.5 Divide and Conquer

Divide and Conquer is an algorithmic paradigm that breaks a problem into smaller subproblems, solves them independently, and then combines their solutions to solve the original problem. It is useful in solving complex problems efficiently by breaking them down into simpler parts.

### 7.5.1 Working of Divide and Conquer Algorithm

1. **Divide:** Split the problem into smaller, more manageable subproblems of the same type. Each subproblem should represent a part of the overall problem. The goal is to divide the problem until no further division is possible.
2. **Conquer:** Solve each subproblem independently (often recursively). If a subproblem is small enough (often referred to as the “base case”), it can be solved directly without further recursion. The goal is to find solutions for these subproblems independently.
3. **Combine:** Merge the results of the subproblems to form a solution for the original problem. Once the smaller subproblems are solved, they can be recursively combined to get the solution of larger problem. The goal is to formulate a solution for the original problem by merging the results from the subproblems.

### 7.5.2 Examples of Divide and Conquer technique

#### 1. Find maximum elements in an array

```
def find_maximum(arr, left, right):
    # Base case: if the array has only one element
```

```

if left == right:
    return arr[left]

# Divide the array into two halves
mid = (left + right) // 2

# Recursively find the maximum in both halves
left_max = find_maximum(arr, left, mid)
right_max = find_maximum(arr, mid + 1, right)

# Return the maximum of both halves
return max(left_max, right_max)

# Example usage
array = [3, 1, 4, 1, 5, 9, 2]
max_element = find_maximum(array, 0, len(array) - 1)
print("Maximum element:", max_element)

# Output: Maximum element: 9

```

## 2. Find the minimum elements in an array

```

def find_minimum(arr, left, right):
    # Base case: if the array has only one element
    if left == right:
        return arr[left]

    # Divide the array into two halves
    mid = (left + right) // 2

    # Recursively find the minimum in both halves
    left_min = find_minimum(arr, left, mid)
    right_min = find_minimum(arr, mid + 1, right)

    # Return the minimum of both halves
    return min(left_min, right_min)

# Example usage
array = [3, 1, 4, 1, 5, 9, 2]

```

```
min_element = find_minimum(array, 0, len(array) - 1)
print("Minimum element:", min_element)

# Output: Minimum element: 1
```

### 3. Merge 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

# Example usage
array = [38, 27, 43, 3, 9, 82, 10]
merge_sort(array)
```



```
print("Sorted array:", array)
```

```
# Output: Sorted array: [3, 9, 10, 27, 38, 43, 82]
```

#### 4. Binary Search

```
def binary_search(arr, left, right, target):  
    if left <= right:  
        mid = (left + right) // 2  
  
        if arr[mid] == target:  
            return mid  
        elif arr[mid] < target:  
            return binary_search(arr, mid + 1, right, target)  
        else:  
            return binary_search(arr, left, mid - 1, target)  
    else:  
        return -1
```

```
# Example usage
```

```
array = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
target = 5
```

```
index = binary_search(array, 0, len(array) - 1, target)
```

```
print("Element found at index:", index)
```

```
# Output: Element found at index: 4
```

#### 5. Quick Sort

```
def quick_sort(arr, low, high):  
    if low < high:  
        pi = partition(arr, low, high)  
  
        quick_sort(arr, low, pi - 1)  
        quick_sort(arr, pi + 1, high)
```

```
def partition(arr, low, high):  
    pivot = arr[high]  
    i = low - 1
```

```

for j in range(low, high):
    if arr[j] <= pivot:
        i += 1
        arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]
return i + 1

# Example usage
array = [10, 7, 8, 9, 1, 5]
quick_sort(array, 0, len(array) - 1)
print("Sorted array:", array)

# Output: Sorted array: [1, 5, 7, 8, 9, 10]

```

### Complexity:

- **Time Complexity:** Dependent on the problem but often logarithmic or linearithmic. For example:
  - Merge Sort:  $O(n \log n)$
  - Quick Sort (average case):  $O(n \log n)$
  - Binary Search:  $O(\log n)$
- **Space Complexity:** Depends on recursion depth. Some algorithms may require extra space, while others use in-place solutions.

### Applications:

- Divide and Conquer technique is used in Merge Sort and Quick Sort for efficient sorting of data.
- Binary Search uses divide and conquer for faster searching in sorted arrays.
- Many dynamic programming solutions, such as the Longest Common Subsequence or Matrix Chain Multiplication, use divide and conquer approaches.

### Benefits:

- Divide and Conquer algorithms often outperform brute-force methods, particularly with large datasets, by reducing the number of operations required.
- This approach is valuable for solving complex problems, such as the Tower of Hanoi, by breaking them into smaller, manageable sub-problems and combining the solutions.
- The nature of Divide and Conquer allows for parallel processing of independent sub-problems, making it well-suited for parallel computing environments.
- Divide and conquer has a recursive nature which is intuitive and easy to implement for many problems.

### Limitations:

- The process of dividing a problem into sub-problems and merging their solutions can introduce additional time and resource overhead, which might be significant for simpler problems.
- Breaking a problem into smaller sub-problems can complicate the overall solution, especially when the sub-problems are interdependent and require specific sequencing.
- Sometimes, the task of combining solutions from sub-problems can be as complex as solving the original problem.
- Deep recursion, which is often used in Divide and Conquer, can lead to stack overflow issues, particularly with large input sizes, due to the extensive memory required for intermediate results.

## 7.6 Fractional Knapsack Problem

### 7.6.1 Problem Statement

#### Problem Statement:

The Fractional Knapsack Problem is a type of optimization problem that involves selecting items to maximize the total value in a knapsack, given a weight constraint. Unlike the 0-1 knapsack problem, where you must either take the entire item or leave it, the fractional knapsack problem allows you to take fractions of an item. This flexibility makes the fractional knapsack problem more straightforward to solve compared to its 0-1 counterpart.

In simple terms, imagine you have a knapsack (or bag) with a limited capacity, and you need to fill it with items to achieve the maximum total value. Each item has a weight and a value, and in the fractional knapsack problem, you can take a part of an item if it's beneficial.

### 7.6.2 Example and Approach

#### Example:

Suppose you have 3 items with weights of 10, 20 and 30 and values as \$60, \$100 and \$120 and the weight limit of the knapsack is 50 units.

#### Approach:

1. Value per unit weight is calculated for each item.
  1. Item 1:  $\$60 / 10 = \$6$  per unit
  2. Item 2:  $\$100 / 20 = \$5$  per unit
  3. Item 3:  $\$120 / 30 = \$4$  per unit
2. Items are sorted based on this value per unit weight in descending order: Item 1, Item 2, Item 3
3. The items are added to the knapsack starting from the highest value per unit weight.
  1. Item 1 can be added fully, the capacity of knapsack used is 10 units and remaining capacity is 40 units. The value added is \$60.
  2. Item 2 can also be added fully, the capacity of knapsack used is 30 units and remaining capacity is 20 units. The value added is \$100.
  3. Item 3 cannot be added fully.
4. If the knapsack cannot fit the entire item, a fraction of it is taken.
  1. Only a fraction of item 3 is added (2/3). Value added is  $(2/3) * \$120 = \$80$  and the total value added is \$240.

### 7.6.3 Difference between 0-1 Knapsack Problem and Fractional Knapsack Problem

- **0-1 Knapsack Problem:** In this problem, you must decide whether to take the entire item or leave it. You cannot take a fraction of an item. This problem is typically solved using dynamic programming.
- **Fractional Knapsack Problem:** Here, you can take fractions of items. It is solved using a greedy approach, which is more efficient than the dynamic programming approach used for the 0-1 knapsack problem.

### 7.6.4 0-1 Knapsack Problem

```
def knapsack_01(weights, values, capacity):  
    n = len(weights)  
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]  
  
    for i in range(1, n + 1):  
        for w in range(capacity + 1):  
            if weights[i-1] <= w:  
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
```

```

        else:
            dp[i][w] = dp[i-1][w]

    return dp[n][capacity]

# Example usage
weights = [1, 2, 3]
values = [60, 100, 120]
capacity = 5
print("Maximum value in 0-1 Knapsack:", knapsack_01(weights, values, capacity))

# Output: Maximum value in 0-1 Knapsack: 220

```

#### Time Complexity:

- $O(n \times W)$ , where  $n$  is the number of items and  $W$  is the capacity of the knapsack.
- The algorithm uses a 2D table (DP table) where the size of the table is  $(n+1) \times (W+1)$ . The extra  $+1$  is because we include 0 as an option for both items and capacities.
- Each cell in this table is filled based on decisions involving previous cells, which results  $O(n \times W)$  operations.

#### Space Complexity:

- $O(n \times W)$
- The DP table requires  $(n+1) \times (W+1)$  space to store the maximum values achievable for each subproblem.

### 7.6.5 Fractional Knapsack Problem

```

def fractional_knapsack(weights, values, capacity):
    n = len(weights)
    items = [(values[i] / weights[i], weights[i], values[i]) for i in range(n)]
    items.sort(reverse=True, key=lambda x: x[0])

    total_value = 0
    for value_per_weight, weight, value in items:
        if capacity <= 0:
            break
        take_weight = min(weight, capacity)
        total_value += take_weight * value_per_weight
        capacity -= take_weight

    return total_value

# Example usage

```

```

weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50

print("Maximum value in Fractional Knapsack:", fractional_knapsack(weights, values, capacity))

# Output: Maximum value in Fractional Knapsack: 240.0

```

#### Time Complexity:

- $O(n \log n)$ , where  $n$  is the number of items.
- The primary computational cost comes from sorting the items by their value-to-weight ratio. Sorting takes  $O(n \log n)$  time.
- After sorting, the algorithm processes each item in linear time  $O(n)$ . Hence, the overall complexity is dominated by the sorting step, which is  $O(n \log n)$ .

#### Space Complexity:

- $O(n)$  for storing items and intermediate values
- The space used is mainly for storing the items and their value-to-weight ratios in a list. This requires  $O(n)$  additional space.
- The space required for sorting is typically handled in-place by the sorting algorithm, so the additional space used for sorting is minimal.

## 7.7 Algorithm Leetcode Problems

### 7.7.1 Problem 1: Two Sum

#### Problem Statement([Link](#)):

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have **exactly one solution**, and you may not use the *same* element twice. You can return the answer in any order.

#### Challenges Faced:

- `nums.index()` was used first to get the index value in output but when the input array contains duplicate values, `nums.index()` returned the first occurrence of a number, which did not lead to the correct solution. Hence, the indexes were stored separately in the starting of the program.
- The brute-force method checks every possible pair of numbers, but have a time complexity of  $O(n^2)$ , which is inefficient for large inputs.
- It was a challenge to return the original indices of the numbers after sorting because sorting alters their positions. Hence a list of tuples is used that stores both the number and its original index before sorting. This allows access to the original indices even after sorting the array.
- As the input array grows larger, time complexity becomes critical. Approaches that have  $O(n^2)$  or even  $O(n \log n)$  time complexities may become too slow.

#### Approach 1(Two Pointer):

- First a list `indexed_nums` was created that stored each number along with its original index.
- This list was sorted based on the values of the numbers, so that the two-pointer technique could be used.
- Starting with two pointers at the beginning and end of the sorted list, the pointers were moved inward based on whether the sum was greater than or less than the target. If I found a match, the original indices of the numbers were returned.

Time Complexity:  $O(n \log n)$

- Sorting the list `indexed_nums` takes  $O(n \log n)$ , where  $n$  is the length of the list.
- The while loop runs in  $O(n)$  time as it scans through the list.
- Overall Time Complexity is  $O(n \log n)$ , dominated by the sorting step.

## Solution:

### 1. Two Sum

Easy Topics Companies Hint

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have **exactly one** solution, and you may not use the same element twice.

You can return the answer in any order.

**Example 1:**

Input: `nums = [2,7,11,15]`, `target = 9`  
Output: `[0,1]`  
Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

**Example 2:**

Input: `nums = [3,2,4]`, `target = 6`  
Output: `[1,2]`

**Example 3:**

Input: `nums = [3,3]`, `target = 6`  
Output: `[0,1]`

**Constraints:**

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- Only one valid answer exists.

```
1 class Solution:
2     def twoSum(self, nums: List[int], target: int) -> List[int]:
3         indexed_nums = [(num, index) for index, num in enumerate(nums)]
4         N = len(nums)
5         indexed_nums.sort(key=lambda x: x[0])
6         left_index = 0
7         right_index = N-1
8         while (left_index < right_index):
9             left_val, left_original_index = indexed_nums[left_index]
10            right_val, right_original_index = indexed_nums[right_index]
11            total = left_val + right_val
12            if total == target:
13                return (left_original_index, right_original_index)
14            elif total > target:
15                right_index -= 1
16            elif total < target:
17                left_index += 1
```

Ln 17, Col 32

Testcase Test Result

Accepted Runtime: 50 ms

Case 1 Case 2 Case 3

Input

nums =  
[3,3]

target =  
6

Output

## Code:

```
def twoSum(nums, target):
    indexed_nums = [(num, index) for index, num in enumerate(nums)]
    N = len(nums)
    indexed_nums.sort(key=lambda x: x[0])
    left_index = 0
    right_index = N-1
    while (left_index < right_index):
        left_val, left_original_index = indexed_nums[left_index]
        right_val, right_original_index = indexed_nums[right_index]
        total = left_val + right_val
        if total == target:
            return (left_original_index, right_original_index)
        elif total > target:
            right_index -= 1
        elif total < target:
            left_index += 1
```

## Example:

```
nums = [2,7,11,15]
target = 9
twoSum(nums, target)    # Output: (0,1)
```

## Approach 2(Dictionary(Hash Map)):

- Hash map was used (`nums_lookup`) to store the numbers and their indices as the list is iterated.
- For each number, the complement was calculated (i.e., `target - nums[i]`) and checked if the complement was already in the dictionary.

- If the complement existed, the indices of the current number and its complement were returned. Otherwise, the current number was added to the dictionary.

Time Complexity:  $O(n)$

- The algorithm iterates through the list once, resulting in  $O(n)$  time complexity.
- Lookup and insertion in the dictionary both take  $O(1)$  on average.

### Solution:

#### 1. Two Sum

**Solved**

Easy Topics Companies Hint

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have **exactly one** solution, and you may not use the same element twice.

You can return the answer in any order.

**Example 1:**

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

**Example 2:**

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

**Example 3:**

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

**Constraints:**

- $2 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- Only one valid answer exists.

```

1 class Solution:
2     def twoSum(self, nums: List[int], target: int) -> List[int]:
3         nums_lookup = {}
4         for i in range(len(nums)):
5             complement = target - nums[i]
6             if complement in nums_lookup:
7                 return (nums_lookup[complement], i)
8             else:
9                 nums_lookup[nums[i]] = i

```

Testcase Test Result

**Accepted** Runtime: 43 ms

Case 1 Case 2 **Case 3**

Input

```
nums = [3,3]
```

target = 6

Output

Code:

```
def twoSum(nums, target):
    nums_lookup = {}
    for i in range(len(nums)):
        complement = target - nums[i]
        if complement in nums_lookup:
            return (nums_lookup[complement], i)
        else:
            nums_lookup[nums[i]] = i
```

Example:

```
nums = [2,7,11,15]
target = 9
twoSum(nums, target)    # Output: (0,1)
```

### 7.7.2 Problem 2: Path Sum

#### Problem Statement([Link](#)):

Given the root of a binary tree and an integer `targetSum`, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals `targetSum`.

A **leaf** is a node with no children.

#### Challenges Faced:

- At the beginning, it's crucial to grasp the concept of traversing from the root to a leaf node, especially what qualifies as a "leaf node". This problem could be solved by Depth-First Search.
- If the base case is not handled properly, the recursion will continue infinitely, causing runtime errors.

- It is important to correctly identify a leaf node — one without children — and only then check if the current path sum equals targetSum.
- Navigating through all potential paths from root to leaf while subtracting node values from the targetSum can be tricky, especially when keeping track of the remaining sum.
- Another challenge is keeping track of the recursive calls and understanding how each function call is working on a new part of the tree without mixing up the path sums.

#### Approach:

- The solution uses Depth-First Search (DFS) to explore every possible root-to-leaf path. At each node, the function reduces the targetSum by the value of the current node.
- If the current node is a leaf (i.e., it has no left or right children) and its value equals the targetSum, algorithm returns True because a valid path is found.
- If the current node is not a leaf, the recursion continues on both the left and right children with the updated targetSum.
- The recursion explores every possible path from the root to the leaf using depth-first search. At each step, the function traverses one path as deep as possible before backtracking to explore other paths.
- The function starts at the root and checks if it's None. If not, it subtracts the node's value from the targetSum. It then checks if the node is a leaf and whether the remaining targetSum equals the node's value. If it does, it returns True.
- If not, it recursively checks the left and right subtrees with the updated targetSum. The function continues to propagate back up the tree, returning True if a valid path is found.
- For example, If the root has value 5 and the targetSum is 22, we call the function for its left and right children with targetSum = 22 - 5 = 17. This process continues down the tree, exploring all paths until a valid path is found or all paths are exhausted.

#### Time Complexity:

- $O(n)$  — In the worst case, we visit each node once, so the time complexity is proportional to the number of nodes  $n$  in the tree.

#### Solution:

##### 112. Path Sum

Easy Topics Companies

Given the `root` of a binary tree and an integer `targetSum`, return `true` if the tree has a **root-to-leaf** path such that adding up all the values along the path equals `targetSum`.

A leaf is a node with no children.

**Example 1:**

Input: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22`  
 Output: `true`  
 Explanation: The root-to-leaf path with the target sum is shown.

```

1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
9         if root is None:
10             return False
11         if root.val == targetSum and root.left is None and root.right is None:
12             return True
13         return self.hasPathSum(root.left, targetSum - root.val) or self.hasPathSum(root.right, targetSum - root.val)

```

Accepted Runtime: 35 ms

Case 1 Case 2 Case 3

Input:

`root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`

`targetSum = 22`

#### Code:

```

def hasPathSum(root, targetSum):
    if root is None:
        return False

    if root.val == targetSum and root.left is None and root.right is None:
        return True

```



```
return self.hasPathSum(root.left, targetSum - root.val) or self.hasPathSum(root.right, targetSum - root.val)
```

Example:

```
root = [1,2,3]
```

```
targetSum = 5
```

```
hasPathSum(root, targetSum) # Output: false
```

### 7.7.3 Problem 3: Binary Tree Paths

#### Problem Statement([Link](#)):

Given the root of a binary tree, return *all root-to-leaf paths in any order*.

A **leaf** is a node with no children.

**Input:** root = [1,2,3,null,5]

**Output:** ["1->2->5", "1->3"]

#### Challenges Faced:

- The problem requires generating all root-to-leaf paths in a binary tree. Understanding how to traverse the tree and construct these paths correctly was a key challenge.
- Dealing with various tree structures, including empty trees, single-node trees, and more complex structures with varying depths and branching, posed challenges in designing a solution.
- Another challenge was properly maintaining and updating the path as the recursion progressed and ensuring that paths are correctly constructed and stored without modifying the path for other recursive calls.
- It is important to ensure that the solution does not generate duplicate paths and correctly reflects all unique root-to-leaf paths.
- After constructing the paths, converting them into the desired format (i.e., strings with node values separated by ->) required careful handling to ensure the output matches the expected format.

#### Approach:

- The `binaryTreePathsUtil` function performs a DFS traversal. It explores each node, appends the node's value to `current_path`, and checks if the node is a leaf. If a leaf is found, the current path is added to `all_paths`. The function then recursively explores the left and right subtrees.
- Depth-First Search (DFS) was used to explore all possible paths from the root to the leaves. By appending the node values to `current_path` as the recursion progresses, the solution constructs each path accurately.
- The `binaryTreePathsUtil` function ensures that all possible paths are explored, regardless of tree complexity.
- The `current_path` list is managed by appending node values and popping them as the recursion unwinds. This approach maintains the correct state of the path during the traversal and avoids interference between different recursive calls.
- By ensuring that each path is constructed and added to `all_paths` only when a leaf node is reached, the solution avoids generating duplicate paths. Each path is built from the root to a unique leaf.
- After exploring both subtrees of a node, the function uses `current_path.pop()` to remove the node's value before returning. This backtracking ensures that the `current_path` list correctly reflects the path being explored in the next recursive call.
- After collecting all paths as lists of node values, the `binaryTreePaths` method converts each path to the desired string format. This involves joining the node values with -> and converting the final list of paths to the output format.

#### Time Complexity:

- $O(n)$  — Each node in the tree is visited once, and the paths are constructed in linear time relative to the number of nodes. The complexity is thus proportional to the number of nodes  $n$  in the tree.

#### Solution:

## 257. Binary Tree Paths

Easy Topics Companies

Given the `root` of a binary tree, return *all root-to-leaf paths in any order*.

A leaf is a node with no children.

**Example 1:**

```

graph TD
    1((1)) --> 2((2))
    1((1)) --> 3((3))
    2((2)) --> 5((5))
    3((3)) --- null3[ ]
    5((5)) --- null5[ ]
  
```

Input: `root = [1,2,3,null,5]`  
 Output: `["1->2->5","1->3"]`

**Example 2:**

Input: `root = [1]`  
 Output: `["1"]`

**Constraints:**

Solved

```

1 # Definition for a binary tree node.
2 class TreeNode:
3     def __init__(self, val=0, left=None, right=None):
4         self.val = val
5         self.left = left
6         self.right = right
7 class Solution:
8     def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
9         paths = self.binaryTreePathsUtil(root, [], [])
10        paths = ['->'.join(map(str,p)) for p in paths]
11        return paths
12
13    def binaryTreePathsUtil(self, node, current_path, all_paths):
14        if node is None:
15            return
16        current_path.append(node.val)
17
18        if node.left is None and node.right is None:
19            all_paths.append(list(current_path))
20        else:
21            self.binaryTreePathsUtil(node.left, current_path, all_paths)
22            self.binaryTreePathsUtil(node.right, current_path, all_paths)
23
24        current_path.pop()
25        return all_paths
  
```

Python3 Auto

Testcase Test Result

Accepted Runtime: 30 ms

Case 1 Case 2

Input

Code:

```

def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
    paths = self.binaryTreePathsUtil(root, [], [])
    paths = ['->'.join(map(str,p)) for p in paths]
    return paths

def binaryTreePathsUtil(self, node, current_path, all_paths):
    if node is None:
        return
    current_path.append(node.val)

    if node.left is None and node.right is None:
        all_paths.append(list(current_path))
    else:
        self.binaryTreePathsUtil(node.left, current_path, all_paths)
        self.binaryTreePathsUtil(node.right, current_path, all_paths)

    current_path.pop()
    return all_paths
  
```

Example:

```

root = [1,2,3,null,5]
binaryTreePaths(root)

# Output: ["1->2->5","1->3"]
  
```

## 7.7.4 Problem 4: Sort an Array

**Problem Statement**([Link](#)):

Given an array of integers `nums`, sort the array in ascending order and return it.  
You must solve the problem **without using any built-in** functions in  $O(n \log n)$  time complexity and with the smallest space complexity possible.

Input: `nums = [5,2,3,1]`

Output: `[1,2,3,5]`

#### Challenges Faced:

- The primary challenge was to select a sorting algorithm that achieves the required  $O(n \log n)$  time complexity and has acceptable space complexity. Comparison-based algorithms like Quick Sort, Merge Sort, and Heap Sort were considered. Merge Sort was chosen for its stable  $O(n \log n)$  performance and predictable space usage.
- Implementing Merge Sort involves recursion, which adds complexity in terms of managing the recursive calls and merging sorted subarrays. Ensuring that the merging process is efficient and correctly handled was a key challenge.
- While Merge Sort requires additional space proportional to the array size ( $O(n)$ ) for merging, it was necessary to balance this against the requirement for  $O(n \log n)$  time complexity.
- The merging step, where two sorted halves are combined into a single sorted array, required careful handling to ensure that the merge operation is done efficiently and correctly.

#### Approach:

- Merge Sort was used to solve this problem. Merge Sort is a divide-and-conquer algorithm. It works by recursively dividing the array into halves until each subarray contains a single element or is empty (which is inherently sorted), and then merging these sorted subarrays to produce the final sorted array.
- Why Merge Sort?
  - Merge Sort has a guaranteed time complexity of  $O(n \log n)$ , making it suitable for the problem constraints.
  - Although Merge Sort requires  $O(n)$  additional space for merging, it ensures stability and predictable performance, which is advantageous in many scenarios.
- The `sortArray` function recursively divides the array and then merges the sorted subarrays.
- The `merge` function handles the merging of two sorted subarrays into a single sorted array.
- Base Case: If the array has one or zero elements, it is already sorted, so return the array as-is.
- Recursive Case: The midpoint of the array is found and then left and right halves of the array are recursively sorted and these two sorted halves are merged into a single sorted array using the `merge` function.
- Merge function:
  - Two pointers, `i` and `j`, are used to traverse the left and right subarrays respectively.
  - The elements from both subarrays are compared and the smaller element is appended to the `sorted_array`.
  - After one of the subarrays is exhausted, any remaining elements from the other subarray are appended directly to the `sorted_array`.

#### Time Complexity:

- $O(n \log n)$  — Merge Sort divides the array into halves ( $\log n$  divisions) and then merges these halves in linear time ( $O(n)$ ). The combination of these operations results in  $O(n \log n)$  time complexity.

#### Solution:

### 912. Sort an Array

Medium Topics Companies

Given an array of integers `nums`, sort the array in ascending order and return it.

You must solve the problem without using any built-in functions in  $O(n \log n)$  time complexity and with the smallest space complexity possible.

**Example 1:**

Input: `nums = [5,2,3,1]`  
Output: `[1,2,3,5]`  
Explanation: After sorting the array, the positions of some numbers are not changed (for example, 2 and 3), while the positions of other numbers are changed (for example, 1 and 5).

**Example 2:**

Input: `nums = [5,1,1,2,0,0]`  
Output: `[0,0,1,1,2,5]`  
Explanation: Note that the values of `nums` are not necessarily unique.

**Constraints:**

- $1 \leq \text{nums.length} \leq 5 \times 10^4$
- $-5 \times 10^4 \leq \text{nums}[i] \leq 5 \times 10^4$

Python3 Auto

```
1 class Solution:
2     def sortArray(self, nums: List[int]) -> List[int]:
3         if len(nums) <= 1:
4             return nums
5         # Divide the array into two halves
6         mid = len(nums) // 2
7         left_half = self.sortArray(nums[:mid])
8         right_half = self.sortArray(nums[mid:])
9
10        # Merge the sorted halves
11        return self.merge(left_half, right_half)
12
13    def merge(self, left: List[int], right: List[int]) -> List[int]:
14        sorted_array = []
15        i = j = 0
16
17        # Merge the two sorted lists
18        while i < len(left) and j < len(right):
19            if left[i] < right[j]:
20                sorted_array.append(left[i])
21                i += 1
22            else:
23                sorted_array.append(right[j])
24                j += 1
25
```

Testcase Test Result

Accepted Runtime: 38 ms

Code:

```
def sortArray(nums):
    if len(nums) <= 1:
        return nums

    # Divide the array into two halves
    mid = len(nums) // 2
    left_half = sortArray(nums[:mid])
    right_half = sortArray(nums[mid:])

    # Merge the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
    sorted_array = []
    i = j = 0

    # Merge the two sorted lists
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_array.append(left[i])
            i += 1
        else:
            sorted_array.append(right[j])
            j += 1

    # Add any remaining elements from both lists
    sorted_array.extend(left[i:])
    sorted_array.extend(right[j:])
```

```
return sorted_array
```

Example:

```
nums = [5,2,3,1]
```

```
sortArray(nums)
```

```
# Output: [1, 2, 3, 5]
```

## 8. Reference

1. [1] "Graph Algorithms - Data Structures and Algorithms," GeeksforGeeks, [Online]. Available: <https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/#22-graph-algorithms>.
2. [2] "Easy Problem Set," LeetCode, [Online]. Available: <https://leetcode.com/problemset/?difficulty=EASY&page=1>.
3. [3] CS50 - "Data Structures," YouTube, [Online]. Available: [https://www.youtube.com/watch?v=3\\_x\\_Fb31NLE&list=PLqM7alHxFySEQDk2MDfbwEdjd2svVJH9p](https://www.youtube.com/watch?v=3_x_Fb31NLE&list=PLqM7alHxFySEQDk2MDfbwEdjd2svVJH9p).
4. [4] Neso Academy - "Graph Theory Full Course," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=R9PTBwOzceo&list=PLBlnK6fEyqRi3-lvwLGzcaquOs5OBTCww>.
5. [5] Neso Academy - "Algorithms Playlist," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=I37kGX-nZEI&list=PLBlnK6fEyqRgWh1emltdMOz8O2m5X3YYn>.
6. [6] CS50 - "Linked Lists," YouTube, [Online]. Available: [https://www.youtube.com/watch?v=ypJwoz\\_SXTo&list=PLqM7alHxFySG6wgjVeEat\\_ouTli0IBQ6D](https://www.youtube.com/watch?v=ypJwoz_SXTo&list=PLqM7alHxFySG6wgjVeEat_ouTli0IBQ6D).
7. [7] CS50 - "Stacks and Queues," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=wWgIAphfn2U&list=PLqM7alHxFySGwXaessYMemAnITqIZdZVE>.
8. [8] "Introduction to Graph Theory," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=LbcJw-iMIoI>.
9. [9] "Dijkstra's Algorithm," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=j0OUwduDOS0>.
10. [10] "Introduction to Trees," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=pcKY4hjDrxk>.
11. [11] "Binary Search Tree," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=oOHrSzA8Nwg>.
12. [12] "AVL Trees," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=ib4BHvr5-Ao>.
13. [13] "Huffman Coding," YouTube, [Online]. Available: <https://www.youtube.com/watch?v=m1p-eWxrt6g>.