**ARRAY**

**1.TWO SUM –LEETCODE**

A screenshot of a computer program

Description automatically generated

To solve the “**Two Sum**” problem with a time complexity less than O(n^2), I have used a hash table (dictionary in Python). By using that approach it has allowed me to check if the complement of the current number (i.e., the number that, when added to the current number, equals the target) exists in constant time, O (1).

**2.BEST TIME TO BUY AND SELL STOCK-LEETCODE**

A screenshot of a computer program

Description automatically generated

To solve the "**Best Time to Buy and Sell Stock**" problem, I used a single-pass approach by tracking the minimum price encountered and calculating the potential profit at each step to update the maximum profit. By using this approach, I have efficiently handled the problem within O(n) time complexity. Also, I have faced challenges while ensuring the logic correctly to update the minimum price and maximum profit simultaneously.

**3.CONTAINS DUPLICATE-LEETCODE**

**A screenshot of a computer program

Description automatically generated**

**T**o solve the **"Contains Duplicate"** problem, I used a set to track encountered elements, returning True if a duplicate is found during iteration, otherwise False. This approach ensures efficient detection of duplicates in O(n) time. A challenge which I faced was managing memory usage, as the set's size grows with the number of unique elements, which could impact performance with large input sizes.

**4.THREESUM(3SUM) -LEETCODE**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

To solve the "**3Sum**" problem, I used a sorted array combined with a two-pointer technique to efficiently find unique triplets that sum up to zero. By Sorting the array, I have managed duplicates, while the two-pointer approach reduced the search space. I have faced a significant challenge which was ensuring the solution handled large input sizes efficiently; For this I have optimized the algorithm to avoid excessive time complexity and duplicated the triplets, by this I have balanced both performance and correctness.

**5.MAXIMUM SUBARRAY-LEETCODE**

A screenshot of a computer program

Description automatically generated

To solve the **maximum subarray** problem, I have implemented Kadane's Algorithm to efficiently find the maximum sum with O(n) time complexity.

The primary challenge which I faced during solving this problem was ensuring the solution that handles large input sizes and correctly tracks the maximum subarray sum, especially when dealing with arrays containing negative numbers.

**6.** [**Search Insert Position**](https://leetcode.com/problems/search-insert-position/)

**A screenshot of a computer program

Description automatically generated**

To solve the problem of inserting the target value in a sorted array, I implemented a binary search algorithm to achieve O(log n) runtime complexity. The main challenge was ensuring the algorithm correctly adjusted the search range and returned the correct insertion index when the target is not found. Binary search will efficiently narrow down the search space, making it ideal for large input sizes.

**LINKED LIST PROBLEMS**

**1.Reverse Linked List**

A screenshot of a computer program

Description automatically generated

To reverse a singly linked list, I implemented both iterative and recursive approaches. The iterative method uses three pointers to reverse the list in O(n) time with constant space, while the recursive method leverages the call stack to reverse the list. It was easy but I faced a small challenge with recursion which was ensuring the base case and pointer adjustments to correctly handled to avoid stack overflow and maintain list integrity

**2. Merge Two Sorted Lists**

<https://leetcode.com/problems/merge-two-sorted-lists/>

A screenshot of a computer program

Description automatically generated

**Approach:**

I started by creating a dummy node that acts as a placeholder for the head of the merged list. This simplifies the process of managing the merged list and handling edge cases.

Using a while loop, I traversed both linked lists simultaneously. By comparing the values of the current nodes of each list, I appended the smaller node to the merged list and advanced the pointer in that list. This ensures the merged list remains sorted.

Once one of the lists is fully traversed, I appended the remaining nodes from the other list to the merged list. This step ensures that no nodes are left out in the final merged list.

One of the main challenges was managing the pointers correctly during the merging process. Incorrectly updating pointers could easily lead to infinite loops or skipped nodes. Additionally, handling cases where one list is longer than the other required careful consideration to ensure that all nodes were included in the merged list. Using a dummy node helped in managing these challenges more effectively, making the code cleaner and easier to understand.

**3. remove-duplicates-from-sorted-list**

<https://leetcode.com/problems/remove-duplicates-from-sorted-list/>

A screenshot of a computer

Description automatically generated

**Approach**

I started by setting the current node to the head of the linked list. This node will be used to traverse the list and check for duplicates.

Using a while loop, I traversed the list as long as current and current.next are not None. For each node, I checked if its value is the same as the next node's value. If they are equal, it indicates a duplicate, and I updated the next pointer of the current node to skip the duplicate node.

If the values are not equal, I simply moved to the next node.

After traversing the entire list and removing duplicates, I returned to the head of the updated list.

One challenge was ensuring the correct updating of pointers to skip duplicate nodes without losing the reference to the rest of the list. Incorrect pointer updates could lead to losing parts of the list or creating infinite loops. Additionally, handling edge cases, such as an empty list or a list with no duplicates, was necessary to ensure robustness.

**4.Remove elements from linked list**

<https://leetcode.com/problems/remove-linked-list-elements/>

A screenshot of a computer

Description automatically generated

**Approach**

**Dummy Node**: I have used a dummy node that points to the original head of the linked list. This helps to handle cases where the head itself needs to be removed, simplifying the removal process.

Also, I have iterated through the linked list starting from the dummy node. By this, for each node, we can check if the next node has the value that needs to be removed.

**Removal**: If the value matches, we adjust the next pointer of the current node to bypass the node with the value val. This effectively removes the node from the list.

Finally, I have returned dummy.next, which points to the head of the modified list after all the required nodes have been removed.

The primary challenge is dealing with edge cases where the head node itself or multiple consecutive nodes need to be removed. Using a dummy node helped me to simplify these scenarios. Additionally, Although the algorithm runs in O(n) time complexity, it was essential to ensure that each step is optimized to handle the maximum constraints effectively. This involved me checking each node only once and making minimal pointer adjustments.