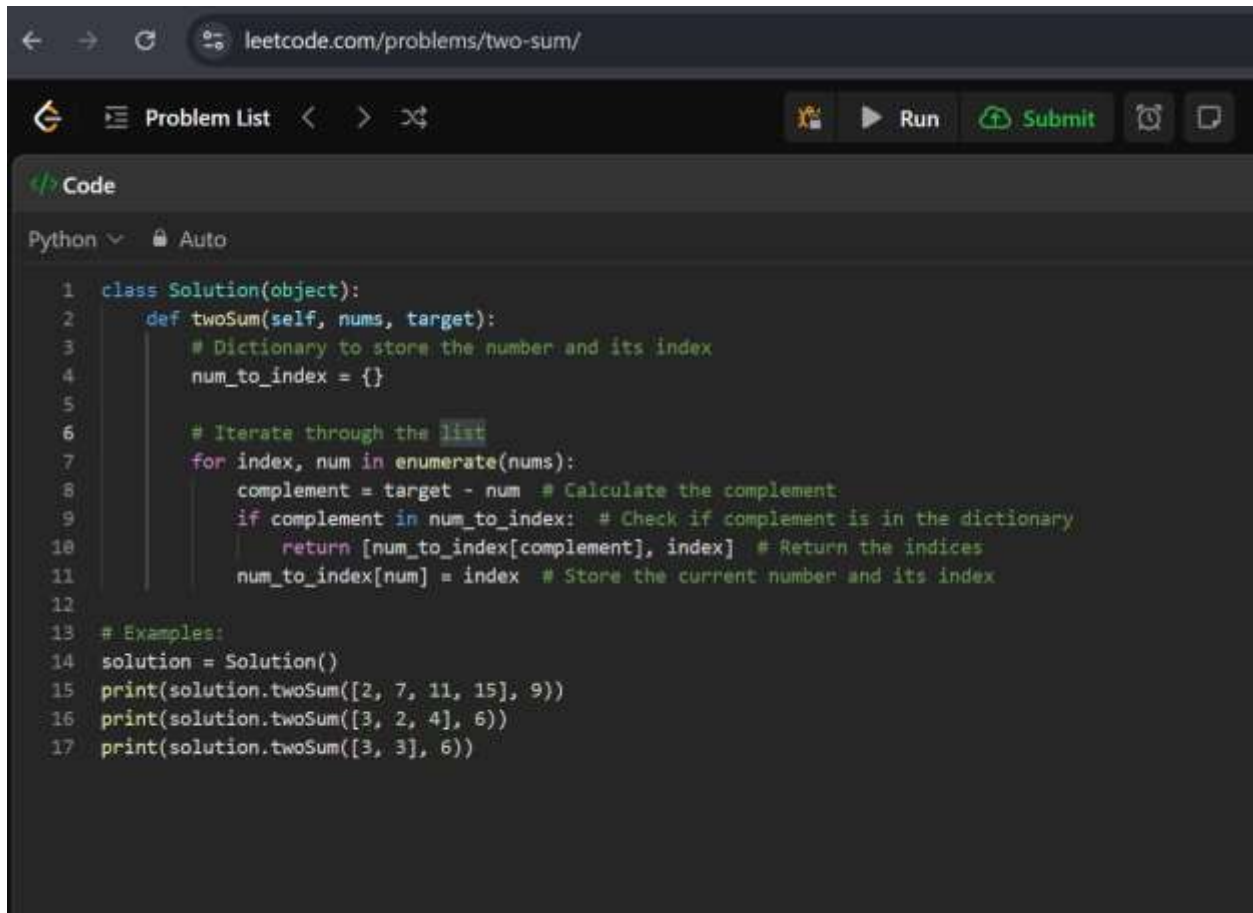


# ARRAY

## 1.TWO SUM –LEETCODE

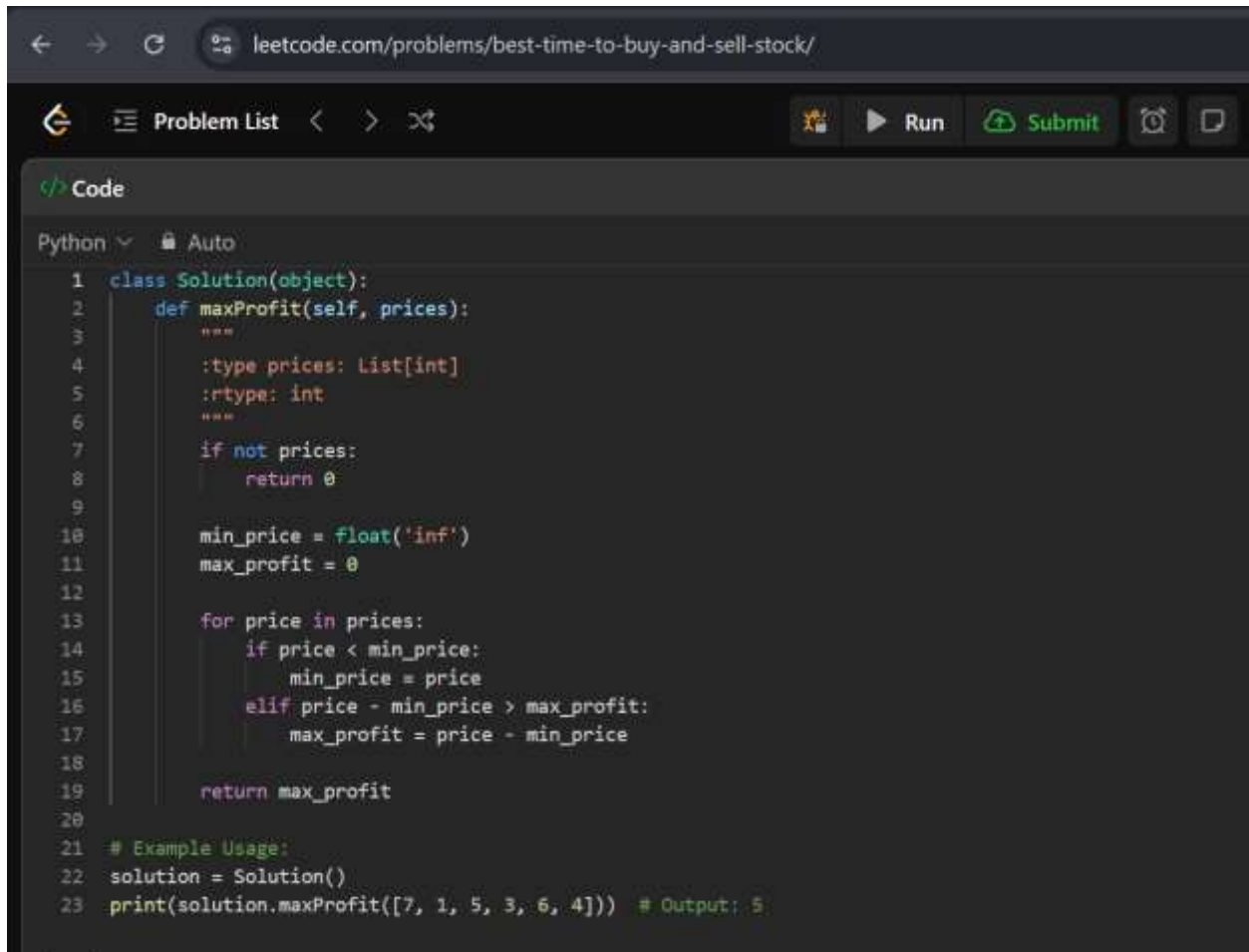


The screenshot shows a web browser at the URL `leetcode.com/problems/two-sum/`. The page displays the 'Two Sum' problem interface. At the top, there are navigation icons and buttons for 'Run', 'Submit', and a timer. Below the problem title, there is a 'Code' editor with a dropdown menu set to 'Python'. The code editor contains the following Python code:

```
1 class Solution(object):
2     def twoSum(self, nums, target):
3         # Dictionary to store the number and its index
4         num_to_index = {}
5
6         # Iterate through the list
7         for index, num in enumerate(nums):
8             complement = target - num # Calculate the complement
9             if complement in num_to_index: # Check if complement is in the dictionary
10                 return [num_to_index[complement], index] # Return the indices
11             num_to_index[num] = index # Store the current number and its index
12
13 # Examples:
14 solution = Solution()
15 print(solution.twoSum([2, 7, 11, 15], 9))
16 print(solution.twoSum([3, 2, 4], 6))
17 print(solution.twoSum([3, 3], 6))
```

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,  $O(1)$ .

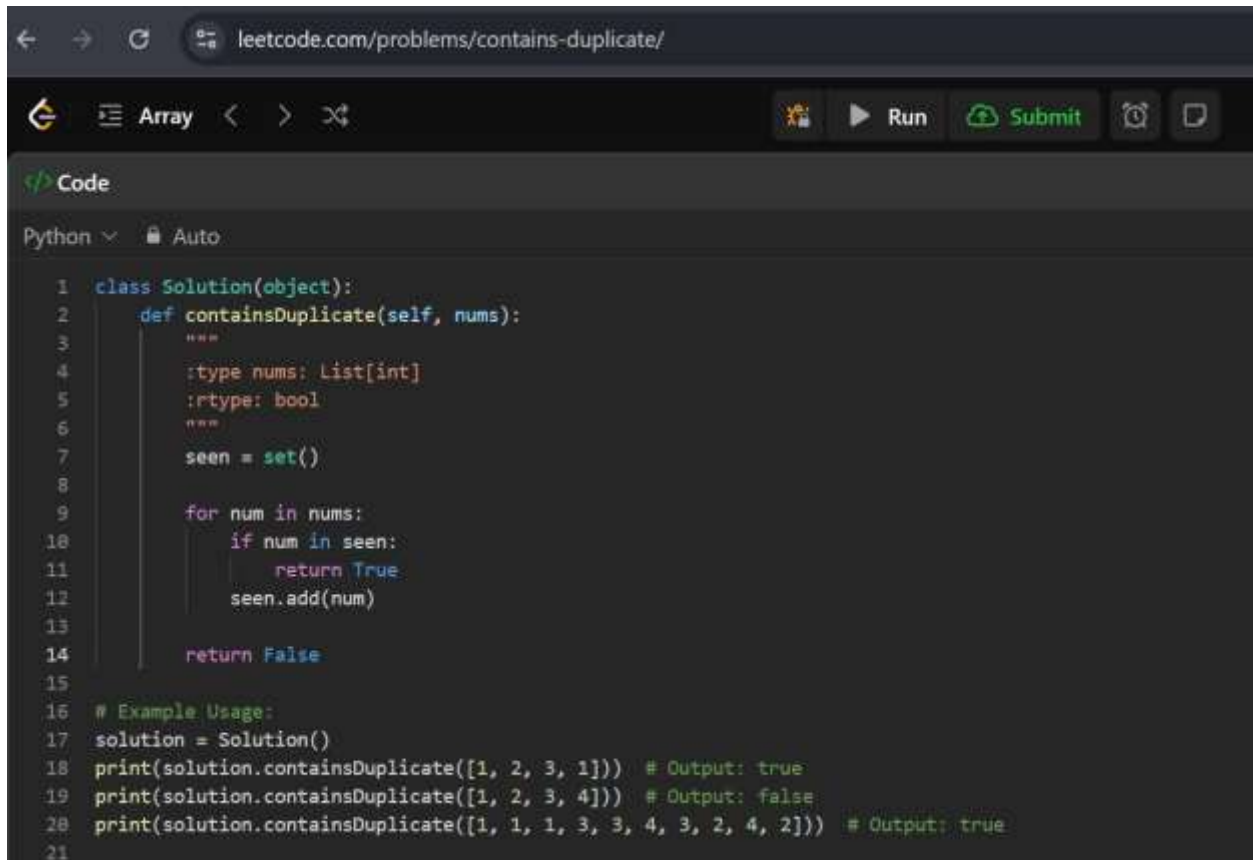
## 2.BEST TIME TO BUY AND SELL STOCK-LEETCODE

The image is a screenshot of a web browser displaying the LeetCode problem page for "Best Time to Buy and Sell Stock". The browser's address bar shows the URL "leetcode.com/problems/best-time-to-buy-and-sell-stock/". The page has a dark theme. At the top, there are navigation icons and buttons for "Run" and "Submit". Below this is a code editor with the title "Code". The code is written in Python and defines a class "Solution" with a method "maxProfit". The method takes a list of prices and returns the maximum profit. The logic involves iterating through the prices, keeping track of the minimum price seen so far, and updating the maximum profit whenever a higher profit is found. An example usage is provided at the bottom of the code block.

```
1 class Solution(object):
2     def maxProfit(self, prices):
3         """
4         :type prices: List[int]
5         :rtype: int
6         """
7         if not prices:
8             return 0
9
10        min_price = float('inf')
11        max_profit = 0
12
13        for price in prices:
14            if price < min_price:
15                min_price = price
16            elif price - min_price > max_profit:
17                max_profit = price - min_price
18
19        return max_profit
20
21 # Example Usage:
22 solution = Solution()
23 print(solution.maxProfit([7, 1, 5, 3, 6, 4])) # Output: 5
```

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



```
1 class Solution(object):
2     def containsDuplicate(self, nums):
3         """
4         :type nums: List[int]
5         :rtype: bool
6         """
7         seen = set()
8
9         for num in nums:
10             if num in seen:
11                 return True
12             seen.add(num)
13
14         return False
15
16 # Example Usage:
17 solution = Solution()
18 print(solution.containsDuplicate([1, 2, 3, 1])) # Output: true
19 print(solution.containsDuplicate([1, 2, 3, 4])) # Output: false
20 print(solution.containsDuplicate([1, 1, 1, 3, 3, 4, 3, 2, 4, 2])) # Output: true
21
```

To 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

```
← → ↺ leetcode.com/problems/3sum/

🏠 Problem List < > 🔍

Code Note X

Python ▾ Auto

1 class Solution(object):
2     def threeSum(self, nums):
3         """
4         :type nums: List[int]
5         :rtype: List[List[int]]
6         """
7         nums.sort() # Sort the array to use two pointers approach
8         result = []
9
10        for i in range(len(nums) - 2):
11            # Skip duplicate values for the fixed element
12            if i > 0 and nums[i] == nums[i - 1]:
13                continue
14
15            # Use two pointers to find the remaining two numbers
16            left, right = i + 1, len(nums) - 1
17            while left < right:
18                total = nums[i] + nums[left] + nums[right]
19                if total < 0:
20                    left += 1
21                elif total > 0:
22                    right -= 1
```

```
← → ↺ leetcode.com/problems/3sum/

🏠 Problem List < > 🔍

Code Note X

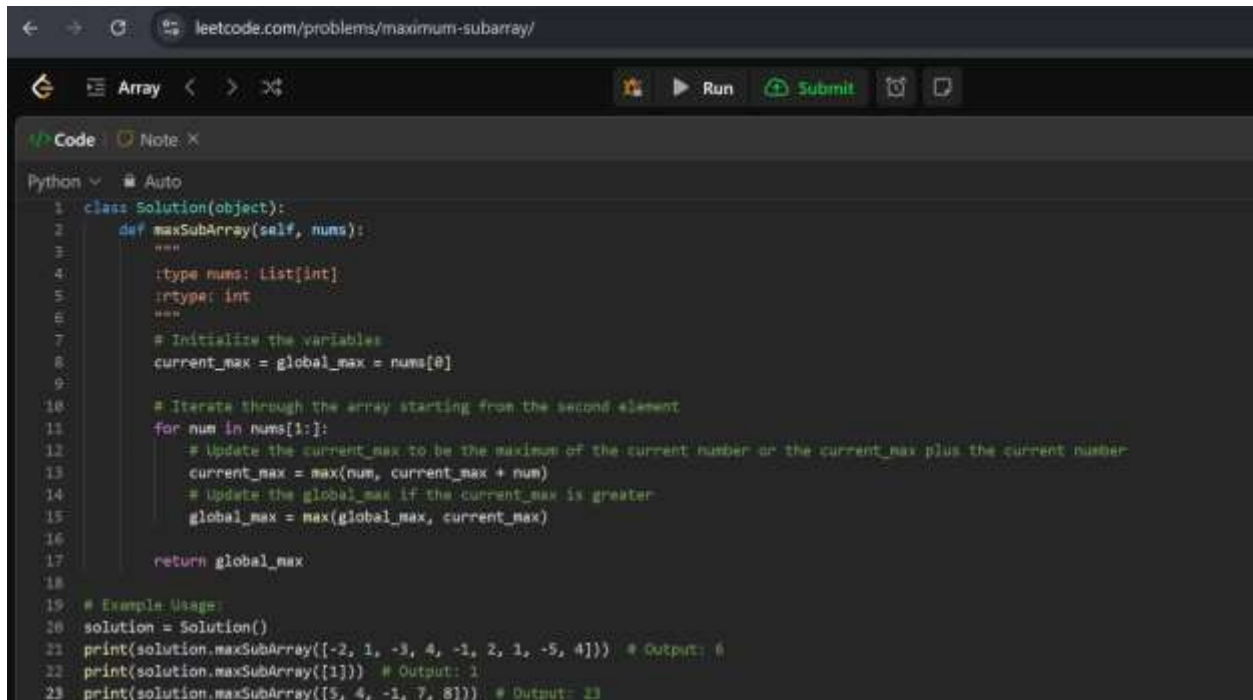
Python ▾ Auto

22            right -= 1
23        else:
24            result.append([nums[i], nums[left], nums[right]])
25            # Skip duplicates for the second and third numbers
26            while left < right and nums[left] == nums[left + 1]:
27                left += 1
28            while left < right and nums[right] == nums[right - 1]:
29                right -= 1
30            left += 1
31            right -= 1
32
33        return result
34
35 # Example Usage:
36 solution = Solution()
37 print(solution.threeSum([-1, 0, 1, 2, -1, -4])) # Output: [[-1, -1, 2], [-1, 0, 1]]
38 print(solution.threeSum([0, 1, 1])) # Output: []
39 print(solution.threeSum([0, 0, 0])) # Output: [[0, 0, 0]]
40
```

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

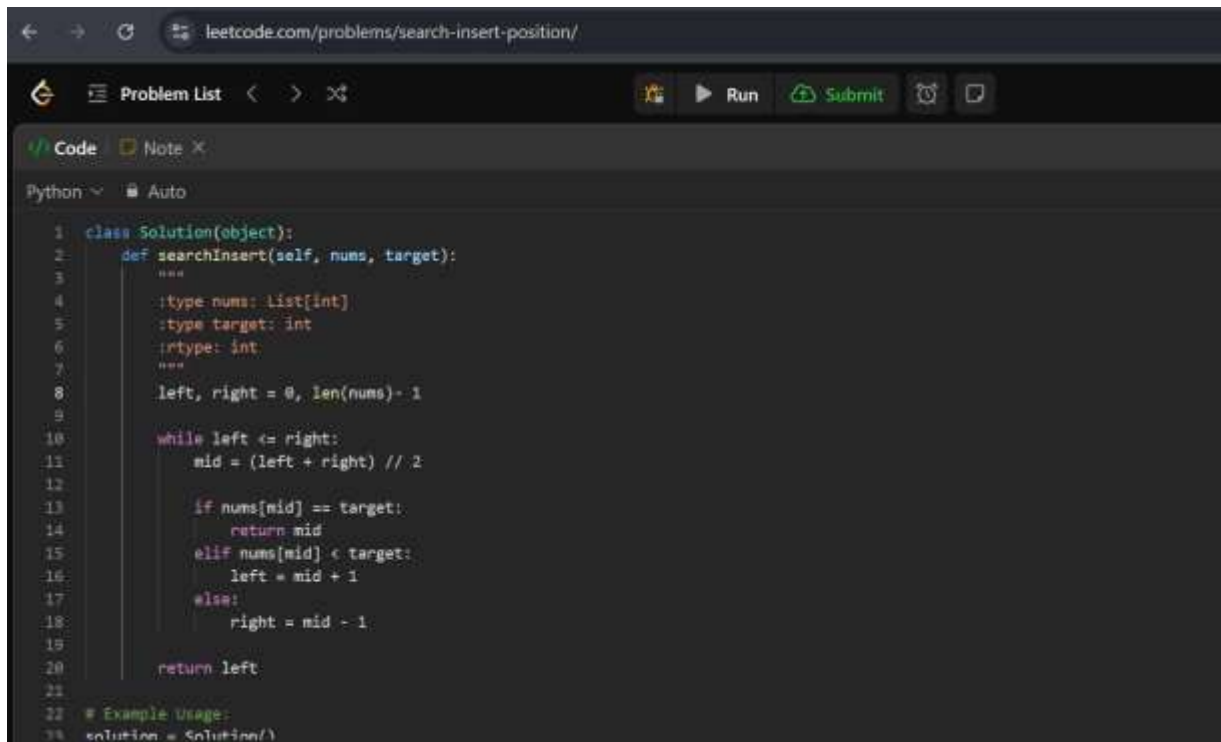
A screenshot of a web browser showing the LeetCode problem page for 'Maximum Subarray'. The browser address bar shows 'leetcode.com/problems/maximum-subarray/'. The page has a dark theme. At the top, there are navigation icons and buttons for 'Run' and 'Submit'. Below the browser window, there is a code editor with a Python tab selected. The code implements Kadane's Algorithm. It defines a class 'Solution' with a method 'maxSubArray'. The method takes a list of integers 'nums' and returns the maximum sum of a contiguous subarray. The code includes type hints, comments, and example usage at the bottom.

```
1 class Solution(object):
2     def maxSubArray(self, nums):
3         """
4         :type nums: List[int]
5         :rtype: int
6         """
7         # Initialize the variables
8         current_max = global_max = nums[0]
9
10        # Iterate through the array starting from the second element
11        for num in nums[1:]:
12            # Update the current_max to be the maximum of the current number or the current_max plus the current number
13            current_max = max(num, current_max + num)
14            # Update the global_max if the current_max is greater
15            global_max = max(global_max, current_max)
16
17        return global_max
18
19 # Example Usage:
20 solution = Solution()
21 print(solution.maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4])) # Output: 6
22 print(solution.maxSubArray([1])) # Output: 1
23 print(solution.maxSubArray([5, 4, -1, 7, 8])) # Output: 23
```

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



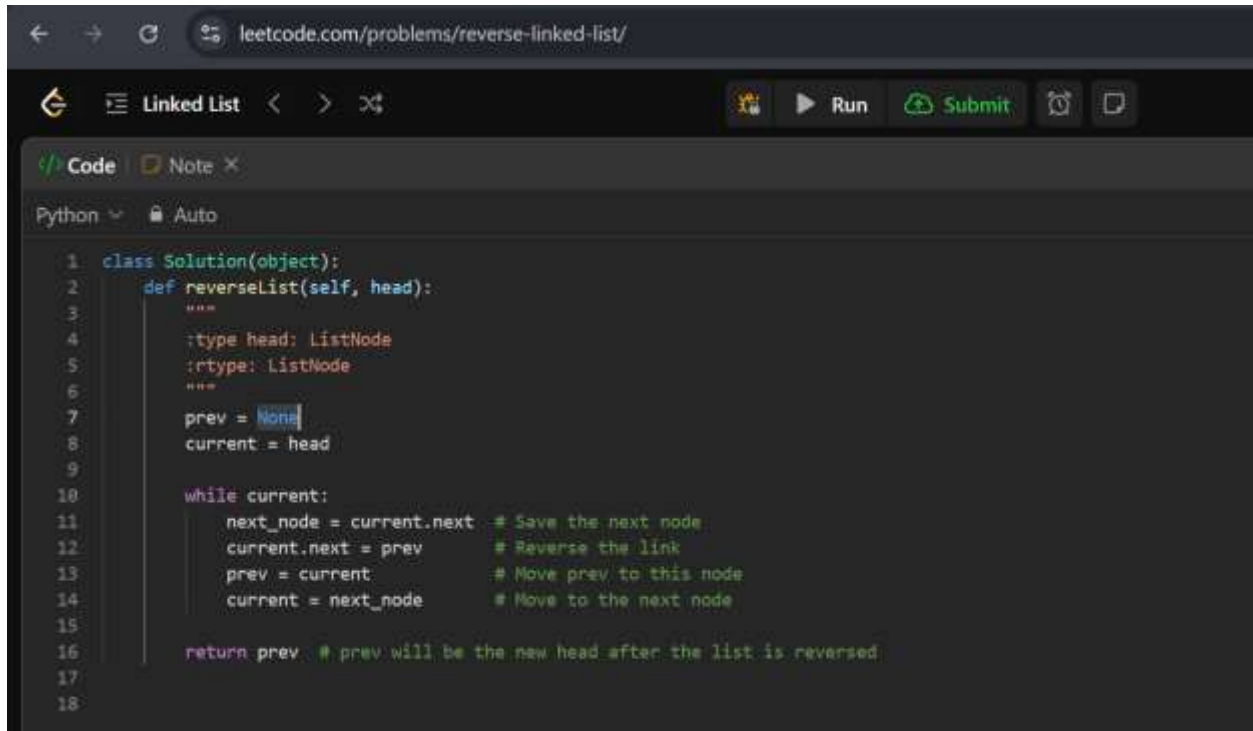
The screenshot shows a web browser window with the URL `leetcode.com/problems/search-insert-position/`. The interface includes a 'Problem List' tab, a 'Run' button, a 'Submit' button, and a 'Code' editor. The code editor is set to 'Python' and 'Auto' mode. The code implements a binary search algorithm to find the insertion position of a target value in a sorted array.

```
1 class Solution(object):
2     def searchInsert(self, nums, target):
3         """
4         :type nums: List[int]
5         :type target: int
6         :rtype: int
7         """
8         left, right = 0, len(nums)-1
9
10        while left <= right:
11            mid = (left + right) // 2
12
13            if nums[mid] == target:
14                return mid
15            elif nums[mid] < target:
16                left = mid + 1
17            else:
18                right = mid - 1
19
20        return left
21
22 # Example Usage:
23 solution = Solution()
```

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

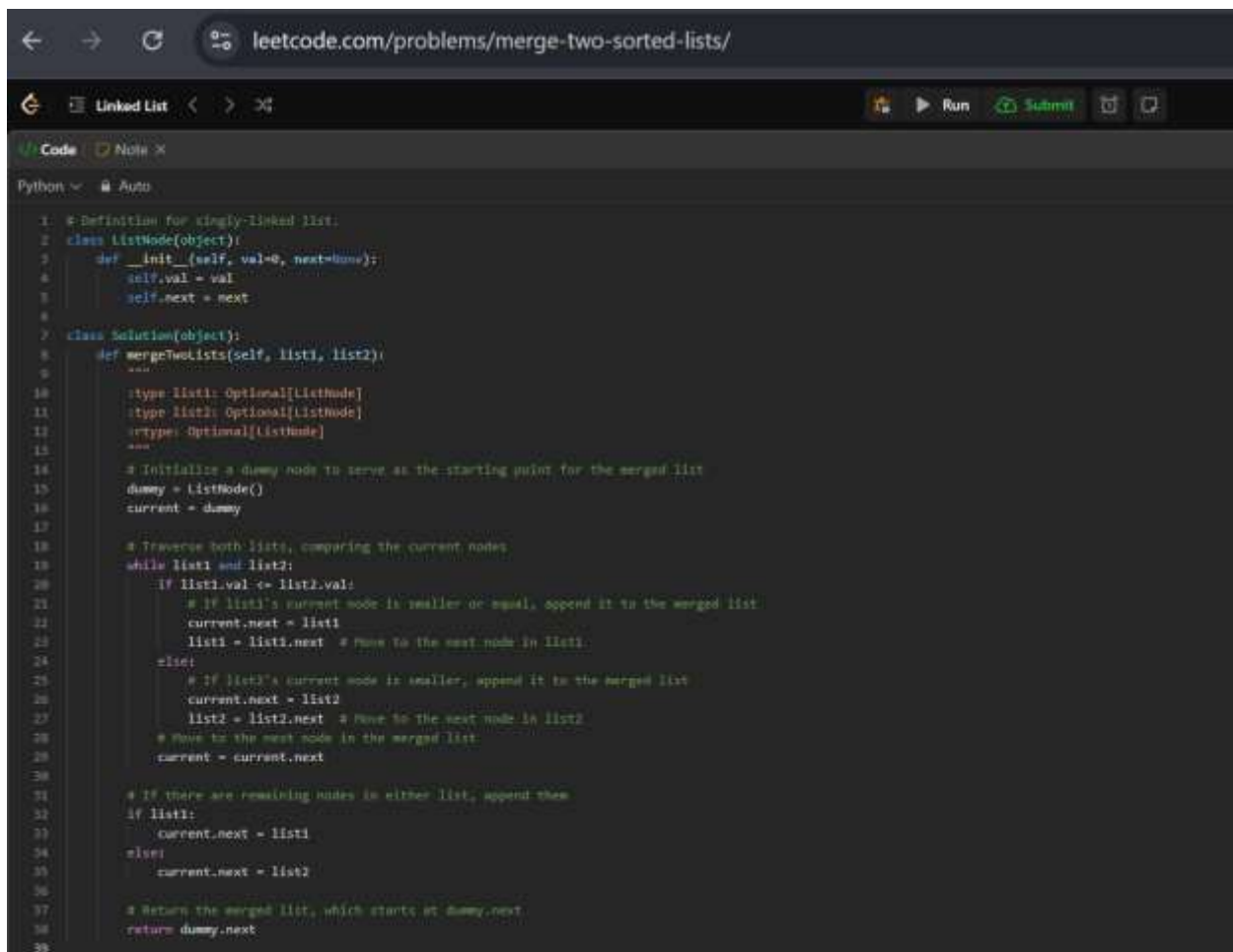
A screenshot of a web browser displaying the LeetCode problem 'Reverse Linked List'. The browser's address bar shows 'leetcode.com/problems/reverse-linked-list/'. The page has a dark theme. At the top, there are navigation icons and a 'Linked List' category label. On the right, there are buttons for 'Run', 'Submit', and a timer. Below the navigation bar, there is a 'Code' editor with a 'Python' language selector and an 'Auto' save indicator. The code editor contains a Python class 'Solution' with a method 'reverseList' that takes 'self' and 'head' as arguments. The method uses an iterative approach to reverse the linked list by maintaining three pointers: 'prev' (initially None), 'current' (initially head), and 'next\_node' (used to store the next node before reversing the link). The code includes comments explaining each step: saving the next node, reversing the link, moving 'prev' to the current node, and moving 'current' to the next node. The method returns 'prev' as the new head of the reversed list.

```
1 class Solution(object):
2     def reverseList(self, head):
3         """
4         :type head: ListNode
5         :rtype: ListNode
6         """
7         prev = None
8         current = head
9
10        while current:
11            next_node = current.next # Save the next node
12            current.next = prev      # Reverse the link
13            prev = current           # Move prev to this node
14            current = next_node      # Move to the next node
15
16        return prev # prev will be the new head after the list is reversed
17
18
```

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/>

The image shows a screenshot of a web browser displaying the LeetCode problem page for 'Merge Two Sorted Lists'. The browser's address bar shows the URL 'leetcode.com/problems/merge-two-sorted-lists/'. Below the browser window, there is a code editor with a dark background. The code is written in Python and defines a 'ListNode' class and a 'Solution' class. The 'Solution' class contains a method 'mergeTwoLists' that takes two linked lists as input and returns a new sorted linked list. The code uses a dummy node to simplify the merging process and a while loop to traverse both lists simultaneously, comparing their values and appending the smaller one to the merged list. After the while loop, any remaining nodes in either list are appended to the merged list. The final return statement returns 'dummy.next', which is the head of the merged list.

```
1 # Definition for singly-linked list.
2 class ListNode(object):
3     def __init__(self, val=0, next=None):
4         self.val = val
5         self.next = next
6
7 class Solution(object):
8     def mergeTwoLists(self, list1, list2):
9         """
10         :type list1: Optional[ListNode]
11         :type list2: Optional[ListNode]
12         :rtype: Optional[ListNode]
13         """
14         # Initialize a dummy node to serve as the starting point for the merged list
15         dummy = ListNode()
16         current = dummy
17
18         # Traverse both lists, comparing the current nodes
19         while list1 and list2:
20             if list1.val <= list2.val:
21                 # If list1's current node is smaller or equal, append it to the merged list
22                 current.next = list1
23                 list1 = list1.next # Move to the next node in list1.
24             else:
25                 # If list2's current node is smaller, append it to the merged list
26                 current.next = list2
27                 list2 = list2.next # Move to the next node in list2
28             # Move to the next node in the merged list
29             current = current.next
30
31         # If there are remaining nodes in either list, append them
32         if list1:
33             current.next = list1
34         else:
35             current.next = list2
36
37         # Return the merged list, which starts at dummy.next
38         return dummy.next
39
```

### 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.

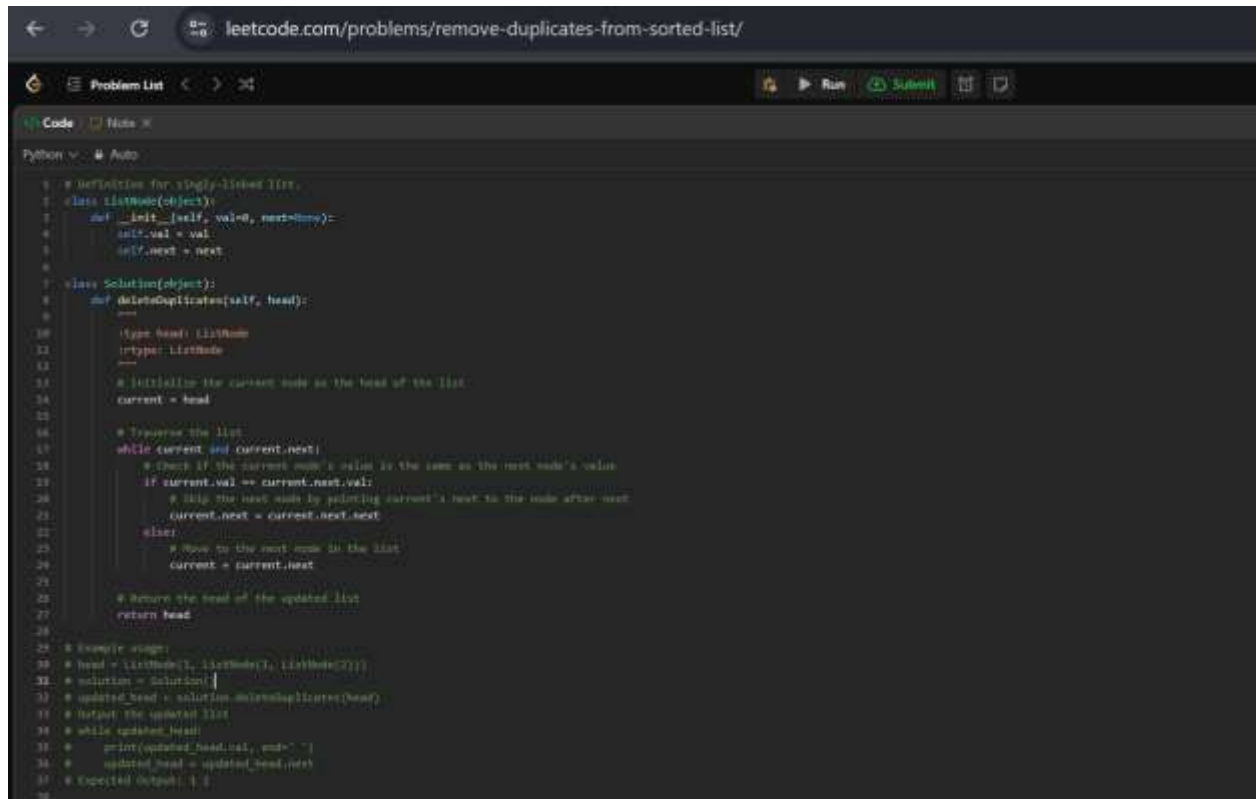
Once one of the lists is fully traversed, I appended the remaining nodes from the other list to the merged list. This step ensures that no nodes are left out in the final merged list.

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 web browser showing the LeetCode problem page for 'remove-duplicates-from-sorted-list'. The browser address bar shows the URL. Below the browser window, there is a code editor with Python code. The code defines a 'ListNode' class and a 'Solution' class with a 'deleteDuplicates' method. The method traverses the linked list, skipping duplicate nodes by updating the 'next' pointer of the current node to 'current.next.next'. The code includes comments in Chinese and English, and a test case at the bottom.

```
1 # Definition for singly-linked list.
2 class ListNode(object):
3     def __init__(self, val=0, next=None):
4         self.val = val
5         self.next = next
6
7 class Solution(object):
8     def deleteDuplicates(self, head):
9         """
10         :type head: ListNode
11         :rtype: ListNode
12         """
13         # Initialize the current node as the head of the list
14         current = head
15
16         # Traverse the list
17         while current and current.next:
18             # Check if the current node's value is the same as the next node's value
19             if current.val == current.next.val:
20                 # Skip the next node by pointing current's next to the node after next
21                 current.next = current.next.next
22             else:
23                 # Move to the next node in the list
24                 current = current.next
25
26         # Return the head of the updated list
27         return head
28
29 # Example usage:
30 # head = ListNode(1, ListNode(1, ListNode(2)))
31 # solution = Solution()
32 # updated_head = solution.deleteDuplicates(head)
33 # Output the updated list
34 # while updated_head:
35 #     print(updated_head.val, end=" ")
36 #     updated_head = updated_head.next
37 # Expected Output: 1 2
```

#### 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.

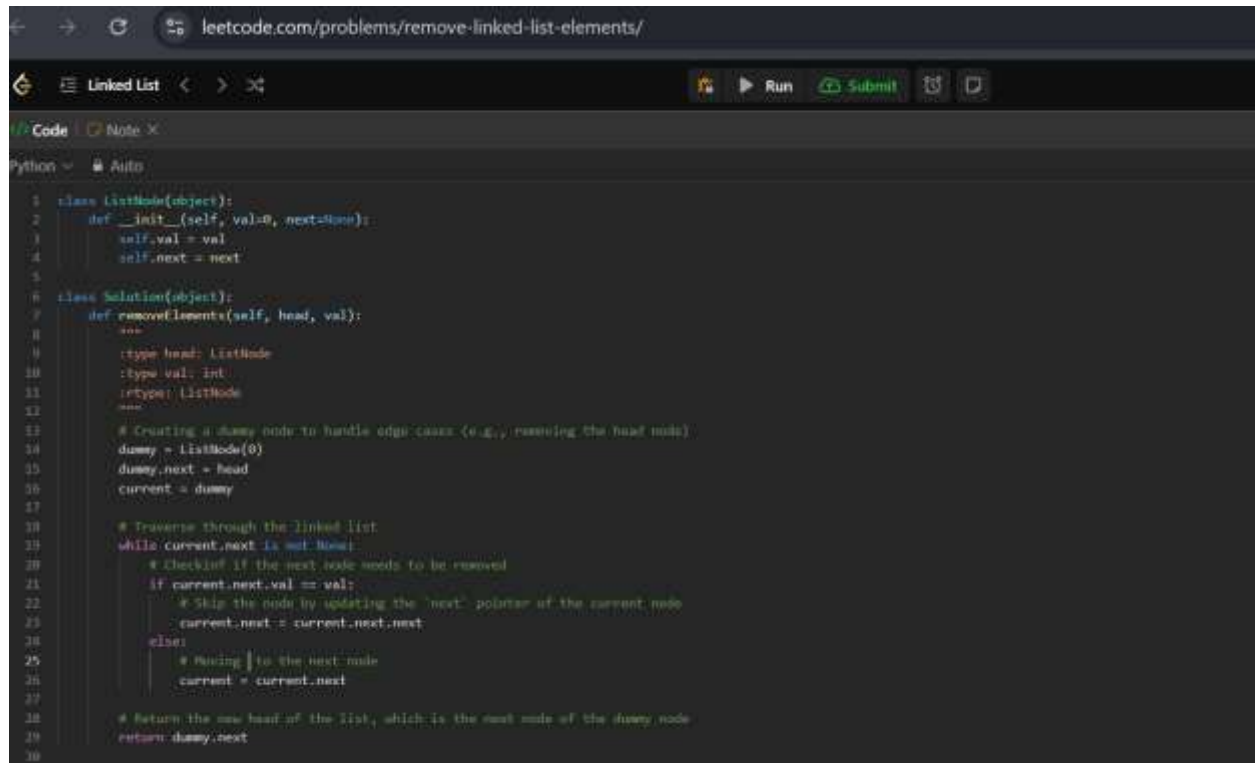
If the values are not equal, I simply moved to the next node.

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 web browser showing the LeetCode problem 'Remove Linked List Elements'. The browser address bar displays the URL 'leetcode.com/problems/remove-linked-list-elements/'. The page header includes the problem title 'Linked List' and navigation icons. The main content area shows a Python code editor with the following code:

```
1 class ListNode(object):
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5
6 class Solution(object):
7     def removeElements(self, head, val):
8         """
9         :type head: ListNode
10        :type val: int
11        :rtype: ListNode
12        """
13        # Creating a dummy node to handle edge cases (e.g., removing the head node)
14        dummy = ListNode(0)
15        dummy.next = head
16        current = dummy
17
18        # Traverse through the linked list
19        while current.next is not None:
20            # Check if the next node needs to be removed
21            if current.next.val == val:
22                # Skip the node by updating the 'next' pointer of the current node
23                current.next = current.next.next
24            else:
25                # Moving to the next node
26                current = current.next
27
28        # Return the new head of the list, which is the next node of the dummy node
29        return dummy.next
30
```

### 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.