Question 1
Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to the target. Return the sum of the three integers.

You may assume that each input would have exactly one solution.

Example 1:
Input: nums = [-1,2,1,-4], target = 1
Output: 2

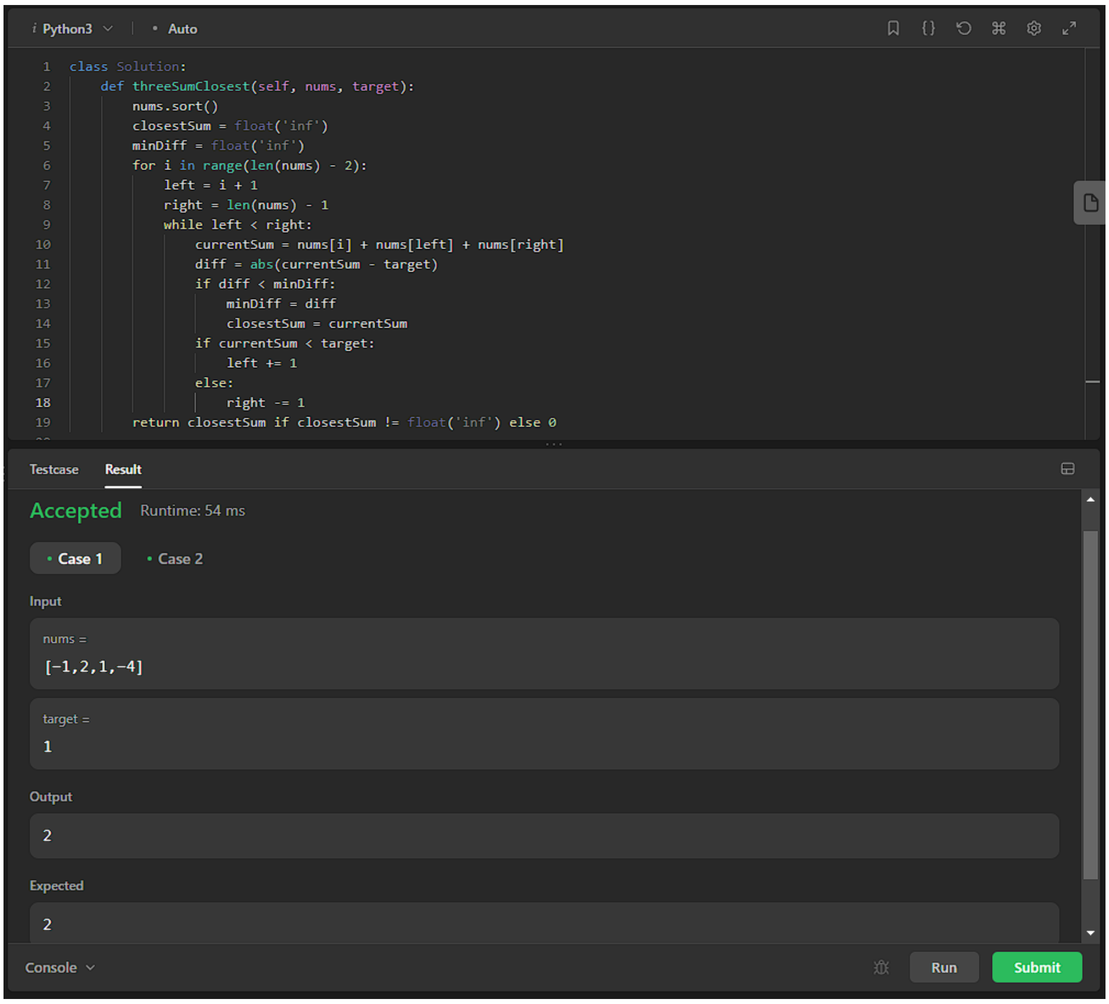
Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

`Approach` :

- Sort the input array nums in ascending order.
- Initialize variables closestSum and minDiff to track the closest sum and minimum difference found so far, respectively. Set both to infinity.
- Iterate over the array nums with the variable i from 0 to len(nums) - 2.
- Within the outer loop, initialize two pointers: left pointing to the element after i and right pointing to the last element of the array.
- While left is less than right, do the following:
    - Calculate the current sum as currentSum = nums[i] + nums[left] + nums[right].
    - Calculate the difference between the current sum and the target: diff = abs(currentSum - target).
    - If the difference is smaller than minDiff, update minDiff to the new difference and update closestSum to the current sum.
    - If the current sum is less than the target, increment left to consider a larger value.
    - If the current sum is greater than the target, decrement right to consider a smaller value.
- After both loops complete, check if closestSum has been updated from its initial value of infinity.
    - If it has been updated, return closestSum as the sum of the three integers closest to the target.
    - If it has not been updated, return 0 (or any default value) to indicate that no valid sum was found.

Time Complexity:O(n)--> It maintains the closest sum and minimum difference found so far, allowing us to achieve O(n) time complexity.
Space Complexity: O(1) --> The space complexity of the algo is O(1), which means it uses constant space regardless of the input size.

In [ ]:
```
def threeSumClosest(nums, target):
    nums.sort()
    closestSum = float('inf')
    minDiff = float('inf')

    for i in range(len(nums) - 2):
        left = i + 1
        right = len(nums) - 1

        while left < right:
            currentSum = nums[i] + nums[left] + nums[right]
            diff = abs(currentSum - target)

            if diff < minDiff:
                minDiff = diff
                closestSum = currentSum

            if currentSum < target:
                left += 1
            else:
                right -= 1

    return closestSum if closestSum != float('inf') else 0
```



```
class Solution:
    def threeSumClosest(self, nums, target):
        nums.sort()
        closestSum = float('inf')
        minDiff = float('inf')
        for i in range(len(nums) - 2):
            left = i + 1
            right = len(nums) - 1
            while left < right:
                currentSum = nums[i] + nums[left] + nums[right]
                diff = abs(currentSum - target)
                if diff < minDiff:
                    minDiff = diff
                    closestSum = currentSum
                if currentSum < target:
                    left += 1
                else:
                    right -= 1
        return closestSum if closestSum != float('inf') else 0
```

Accepted Runtime: 54 ms

Case 1 Case 2

Input

nums =

[-1,2,1,-4]

target =

1

Output

2

Expected

2

Question 2
Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that:

```
• 0 <= a, b, c, d < n
• a, b, c, and d are distinct.
• nums[a] + nums[b] + nums[c] + nums[d] == target
```

You may return the answer in any order.

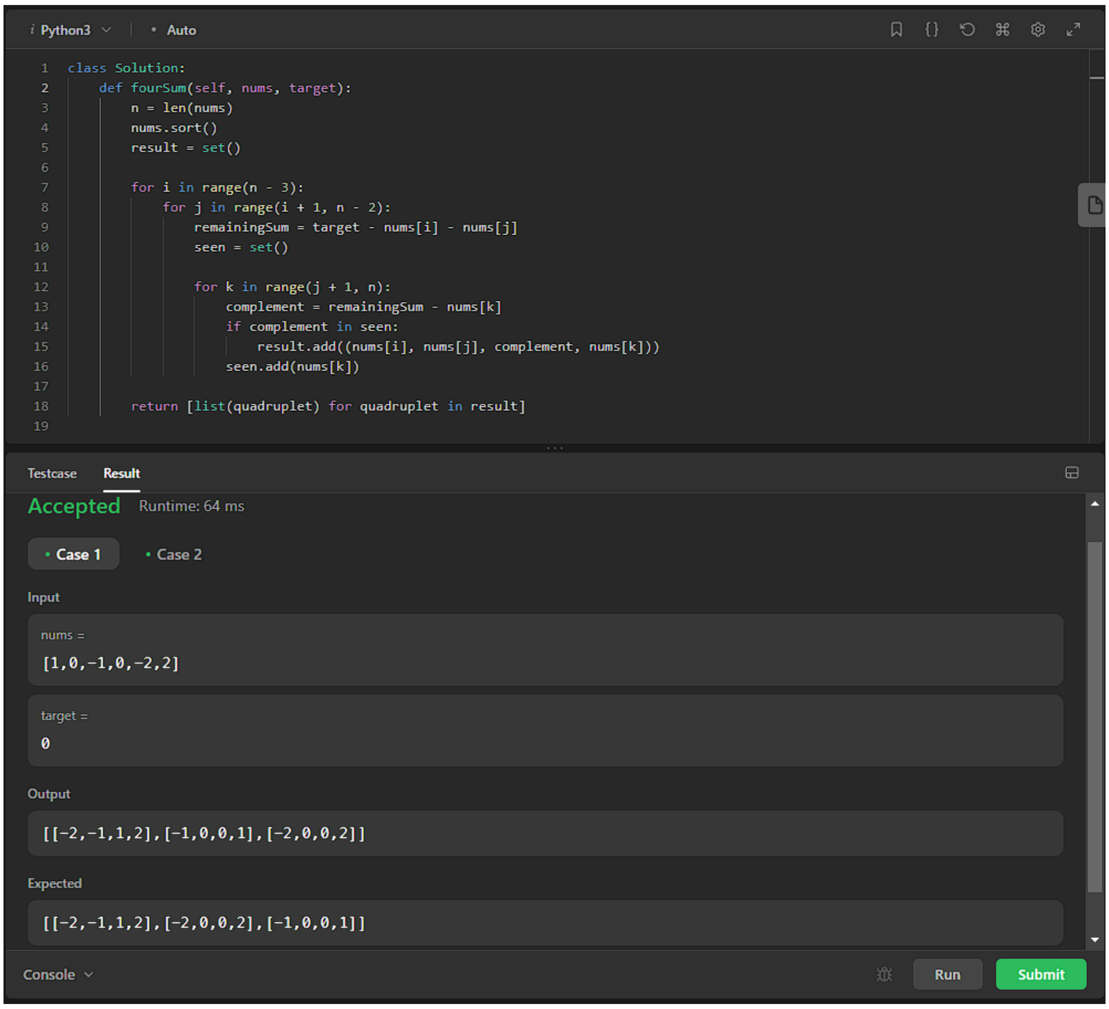
Example 1:
Input: nums = [1,0,-1,0,-2,2], target = 0
Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

`Approach` :

- Sort the input array nums in ascending order.
- Initialize an empty list result to store the unique quadruplets.
- Iterate over the array nums with the variable i from 0 to n-4, where n is the length of nums.
- Within the outer loop, iterate over the array nums with the variable j from i+1 to n-3.
- Within the nested loops, initialize two pointers: left pointing to j+1 and right pointing to n-1.
- While left is less than right, do the following:
  - Calculate the current sum as currentSum = nums[i] + nums[j] + nums[left] + nums[right].
  - If the current sum is equal to the target, append the quadruplet [nums[i], nums[j], nums[left], nums[right]] to the result list.
  - If the current sum is less than the target, increment left to consider a larger value.
  - If the current sum is greater than the target, decrement right to consider a smaller value.
- After the nested loop ends, increment j while skipping duplicates.
- After the outer loop ends, increment i while skipping duplicates.
- Finally, return the result list containing all the unique quadruplets.

Time Complexity O(n^3) --> The time complexity of this algorithm is O(n^3) because there are three nested loops, and each loop can iterate up to n times.
Space Complexity O(1) --> The space complexity of the algorithm is O(1) or constant space.

In [ ]:
```python
def fourSum(nums, target):
    n = len(nums)
    nums.sort()
    result = set()

    for i in range(n - 3):
        for j in range(i + 1, n - 2):
            remainingSum = target - nums[i] - nums[j]
            seen = set()

            for k in range(j + 1, n):
                complement = remainingSum - nums[k]
                if complement in seen:
                    result.add((nums[i], nums[j], complement, nums[k]))
                seen.add(nums[k])

    return [list(quadruplet) for quadruplet in result]
```



```python
class Solution:
    def fourSum(self, nums, target):
        n = len(nums)
        nums.sort()
        result = set()

        for i in range(n - 3):
            for j in range(i + 1, n - 2):
                remainingSum = target - nums[i] - nums[j]
                seen = set()

                for k in range(j + 1, n):
                    complement = remainingSum - nums[k]
                    if complement in seen:
                        result.add((nums[i], nums[j], complement, nums[k]))
                    seen.add(nums[k])

        return [list(quadruplet) for quadruplet in result]
```

Accepted Runtime: 64 ms

Case 1 Case 2

Input

```
nums =
[1,0,-1,0,-2,2]
```

```
target =
0
```

Output

```
[[-2,-1,1,2],[-1,0,0,1],[-2,0,0,2]]
```

Expected

```
[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

Question 3

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container.

If such an arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

```
    • For example, the next permutation of arr = [1,2,3] is
[1,3,2].
    • Similarly, the next permutation of arr = [2,3,1] is
[3,1,2].
    • While the next permutation of arr = [3,2,1] is [1,2,3]
because [3,2,1] does not have a lexicographical larger
rearrangement.
```

Given an array of integers nums, find the next permutation of nums. The replacement must be in place and use only constant extra memory.

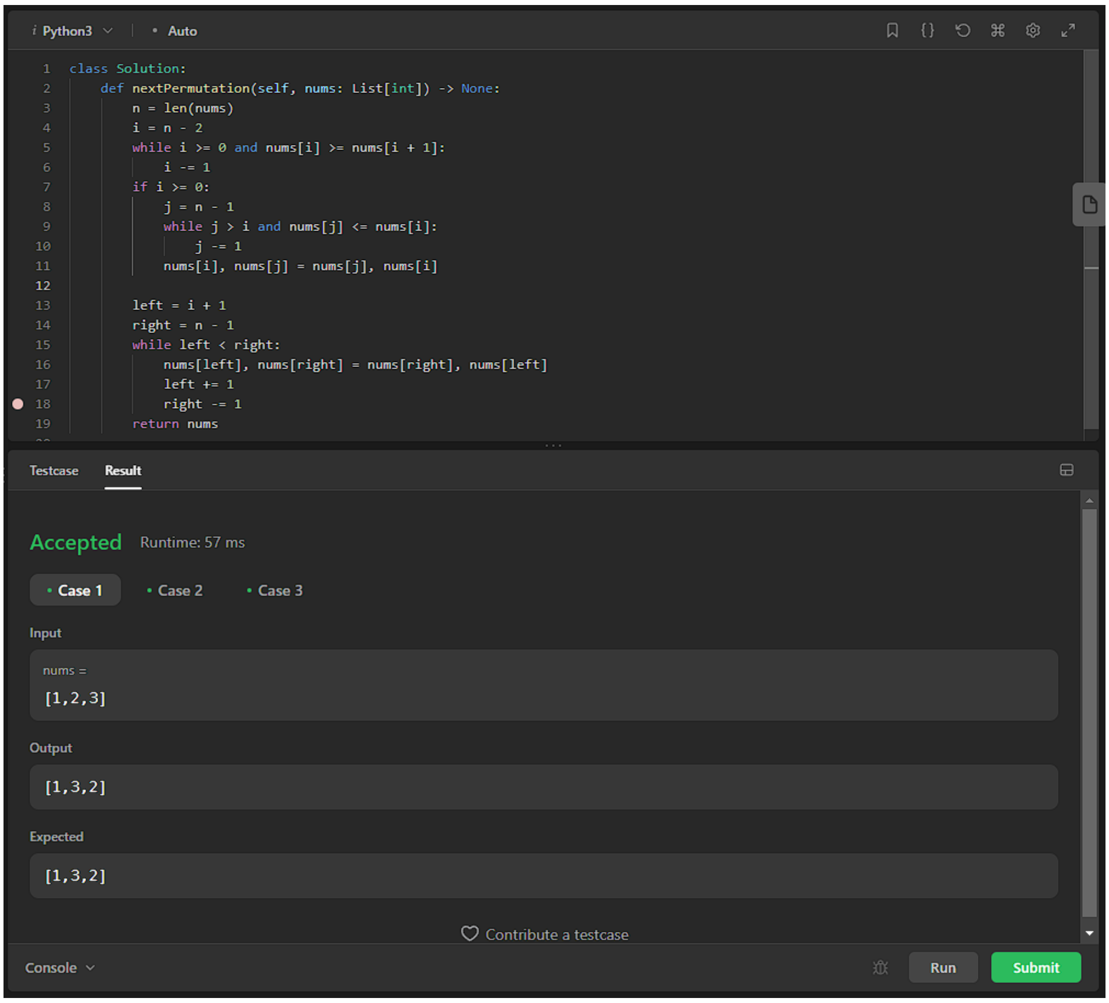
Example 1:
Input: nums = [1,2,3]
Output: [1,3,2]

`Approach` :

- Start from the rightmost element of the array and traverse towards the left until we find a decreasing element. Let's call this element pivot. This step helps us identify the rightmost element that can be modified to create the next permutation.
- If we do not find a decreasing element (i.e., the array is in descending order), it means this is the last permutation. In this case, we reverse the entire array to get the lowest possible order, and we return the modified array.
- If we find a decreasing element, we need to find the next greater element to the right of pivot that is larger than the pivot. We swap these two elements.
- After the swap, we have the rightmost elements in a non-decreasing order. We need to reverse this portion of the array to make it the smallest possible order.
- Finally, we return the modified array.

Time Complexity O(n) --> the overall time complexity of the algorithm is O(n).
Space Complexity O(1) --> The space complexity of the algorithm is O(1) or constant space.

In [ ]:
```
def nextPermutation(nums):
    n = len(nums)
    i = n - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1

    if i >= 0:
        j = n - 1
        while j > i and nums[j] <= nums[i]:
            j -= 1
        nums[i], nums[j] = nums[j], nums[i]

    left = i + 1
    right = n - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1
    return nums
```



```
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        n = len(nums)
        i = n - 2
        while i >= 0 and nums[i] >= nums[i + 1]:
            i -= 1
        if i >= 0:
            j = n - 1
            while j > i and nums[j] <= nums[i]:
                j -= 1
            nums[i], nums[j] = nums[j], nums[i]

        left = i + 1
        right = n - 1
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1
        return nums
```

Accepted Runtime: 57 ms

Case 1 Case 2 Case 3

Input

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

Output

```
[1,3,2]
```

Expected

```
[1,3,2]
```

Question 4 Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

Input: nums = [1,3,5,6], target = 5

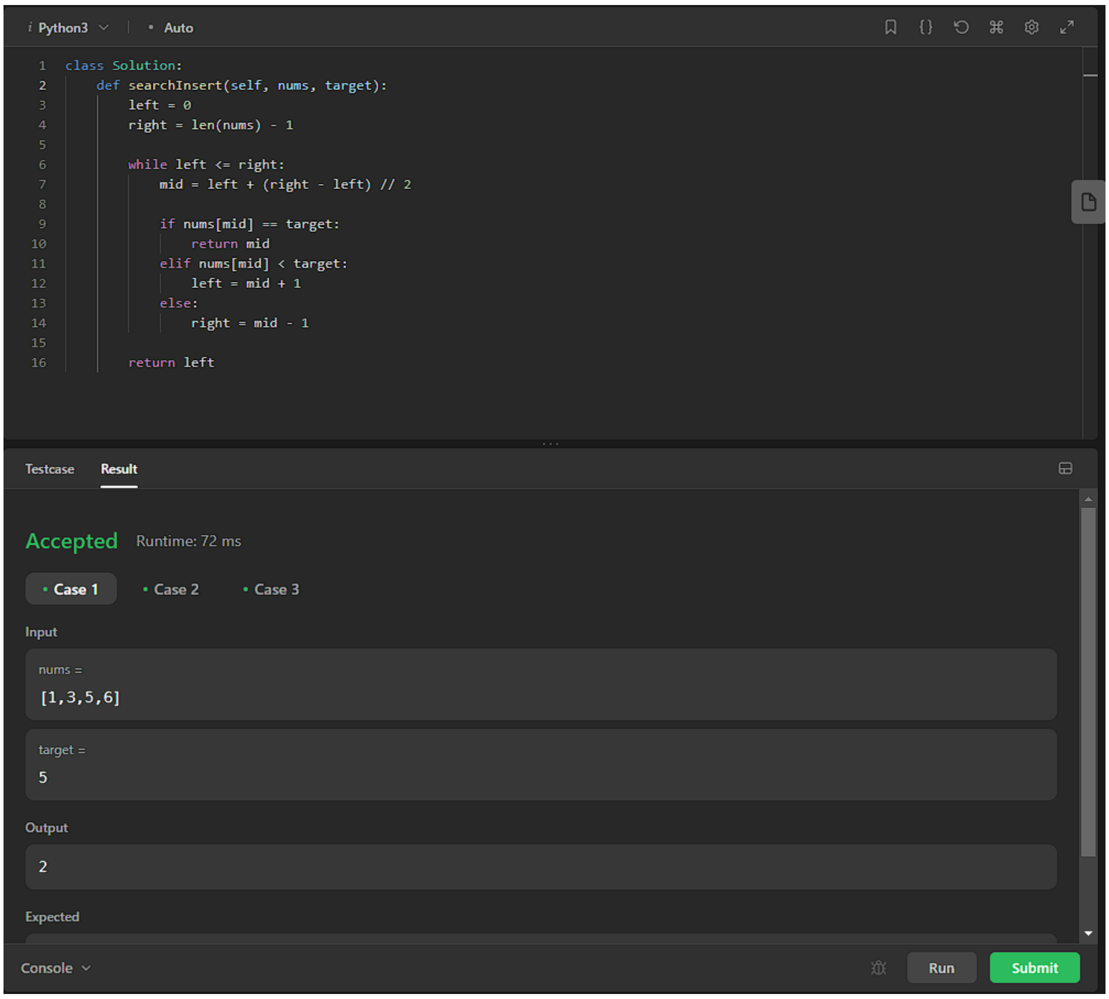
Output: 2

`Approach` :

- Set the left pointer left to 0 and the right pointer right to the length of the array minus 1.
- While left is less than or equal to right, do the following:
    - Calculate the middle index as mid = left + (right - left) // 2.
    - If the value at the middle index is equal to the target, return the middle index.
    - If the value at the middle index is less than the target, update left = mid + 1.
    - If the value at the middle index is greater than the target, update right = mid - 1.
- If the target value is not found in the array, return left.

Time Complexity O(log n) --> The binary search algorithm has a time complexity of O(log n), which satisfies the requirement.
Space Complexity O(1) --> Here insertion index in a sorted array is O(1), or constant space.

In [ ]:
```
def searchInsert(nums, target):
    left = 0
    right = len(nums) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return left
```



```python
class Solution:
    def searchInsert(self, nums, target):
        left = 0
        right = len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1

        return left
```

Accepted Runtime: 72 ms

Case 1 Case 2 Case 3

Input

nums =
[1,3,5,6]

target =
5

Output

2

Expected

Question 5

You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return the resulting array of digits.

Example 1:

Input: digits = [1,2,3]
Output: [1,2,4]

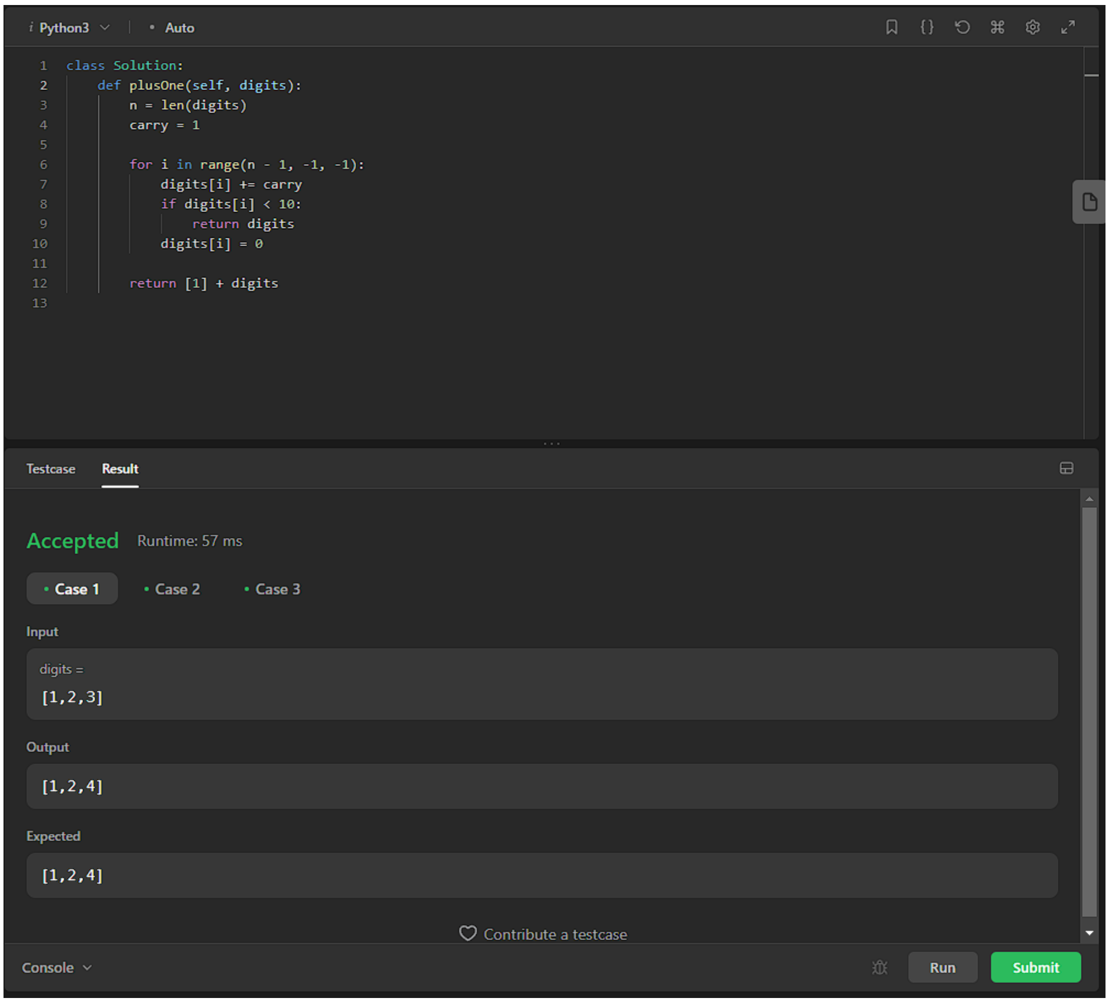
Explanation: The array represents the integer 123.
Incrementing by one gives 123 + 1 = 124.
Thus, the result should be [1,2,4].

`Approach` :

- Start from the least significant digit (the last element in the array) and move towards the most significant digit (the first element in the array).
- Increment the value of the current digit by one.
- If the incremented value is less than 10, we have the result, and we can return the modified array of digits.
- If the incremented value is 10, set the current digit to 0 and move to the next digit.
- Repeat steps 2 to 4 until we reach the most significant digit or we find a digit that does not need to be reset to 0.
- If we reach the most significant digit and its incremented value is 10, we need to add an additional digit 1 at the beginning of the array to accommodate the carry.
- Return the modified array of digits.

Time Complexity O(n) --> Time Complexity for this algo is O(n), where n in the len of array.
Space Complexity O(1) --> Space complexity is O(1).

In [ ]:
```python
def plusOne(digits):
    n = len(digits)
    carry = 1

    for i in range(n - 1, -1, -1):
        digits[i] += carry
        if digits[i] < 10:
            return digits
        digits[i] = 0

    return [1] + digits
```



```
class Solution:
    def plusOne(self, digits):
        n = len(digits)
        carry = 1

        for i in range(n - 1, -1, -1):
            digits[i] += carry
            if digits[i] < 10:
                return digits
            digits[i] = 0

        return [1] + digits

```

Accepted Runtime: 57 ms

Input

digits =

```
[1,2,3]
```

Output

```
[1,2,4]
```

Expected

```
[1,2,4]
```

Question 6

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.

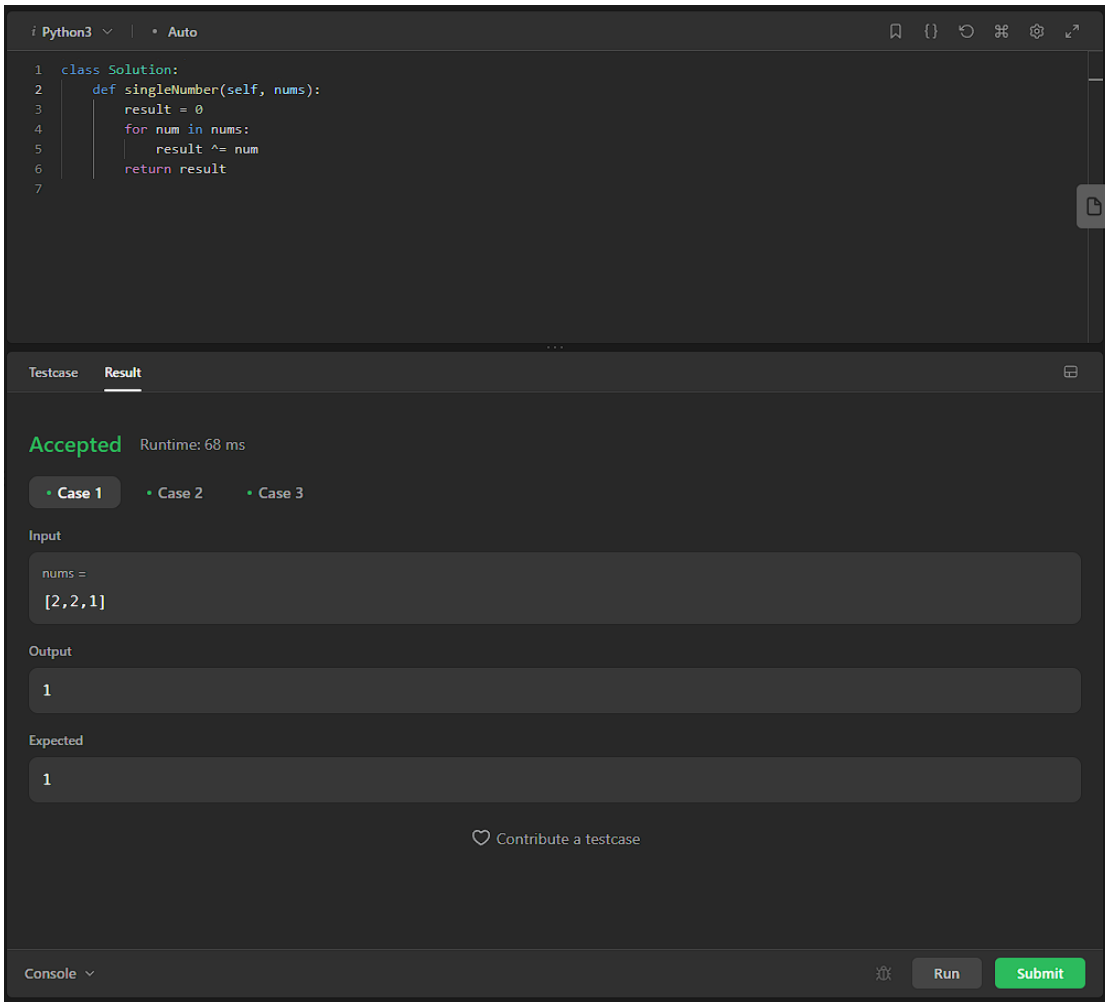
Example 1:
Input: nums = [2,2,1]
Output: 1

`Approach` :

- XORing a number with itself results in zero: a XOR a = 0.
- XORing a number with zero gives the number itself: a XOR 0 = a.
- The XOR operation is commutative: a XOR b = b XOR a.

Time Complexity O(n) --> This algo take O(n) time.
Space Complexity O(1) --> This algo take constant space.

```
In [ ]: def singleNumber(nums):
            result = 0
            for num in nums:
                result ^= num
            return result
```



```
class Solution:
    def singleNumber(self, nums):
        result = 0
        for num in nums:
            result ^= num
        return result
```

Accepted Runtime: 68 ms

Case 1 Case 2 Case 3

Input

nums =
[2,2,1]

Output

1

Expected

1

Question 7

You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within the inclusive range.

A number x is considered missing if x is in the range [lower, upper] and x is not in nums.

Return the shortest sorted list of ranges that exactly covers all the missing numbers. That is, no element of nums is included in any of the ranges, and each missing number is covered by one of the ranges.

Example 1:
Input: nums = [0,1,3,50,75], lower = 0, upper = 99
Output: [[2,2],[4,49],[51,74],[76,99]]

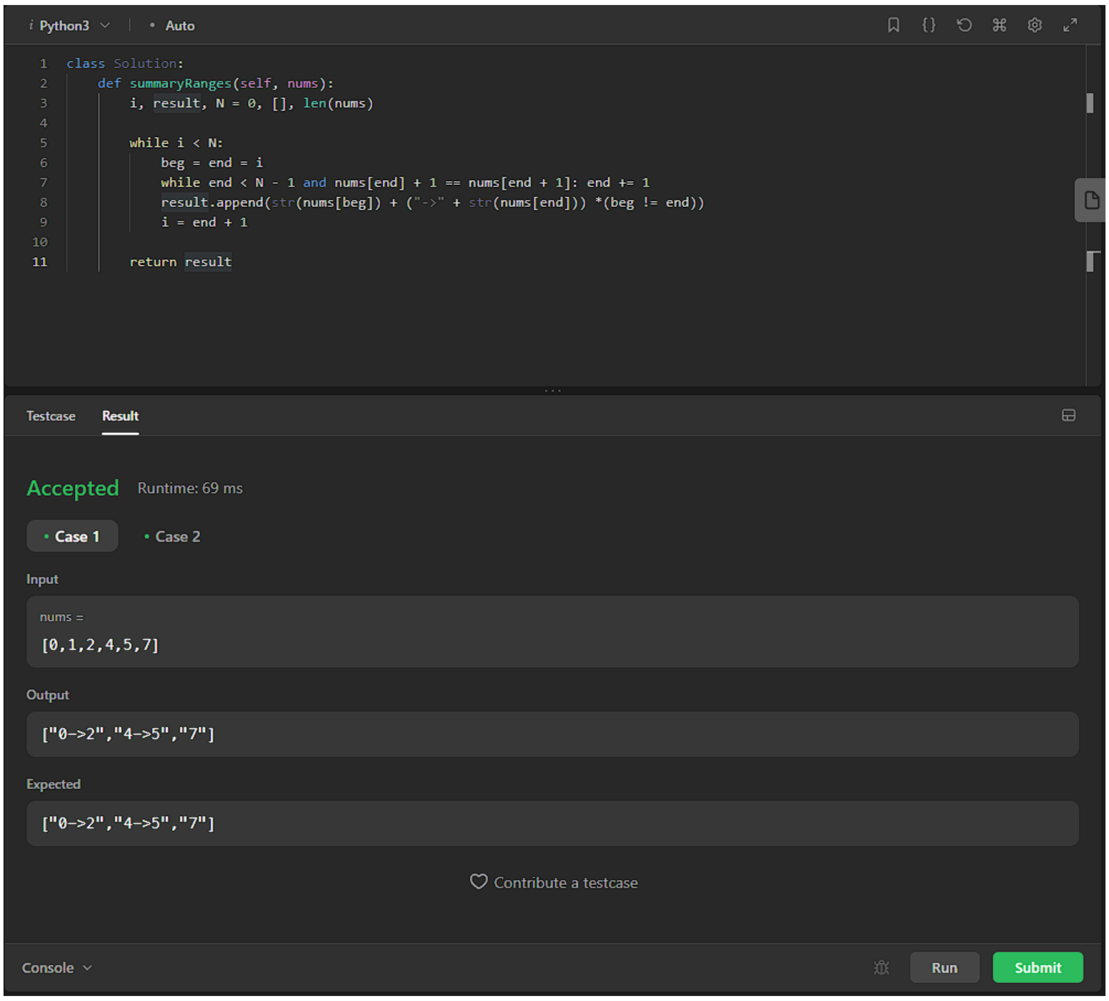
Explanation: The ranges are:
[2,2]
[4,49]
[51,74]
[76,99]

`Approach` :

- Define a function summaryRanges that takes a single parameter nums.
- Initialize variables i and result to 0 and an empty list, respectively.
- Get the length of the input array nums and assign it to N.
- Start a while loop with the condition i < N.
- Inside the while loop:
  - Initialize beg and end to i, representing the beginning and ending indices of a potential range.
  - Start another while loop with the condition end < N - 1 and nums[end] + 1 == nums[end + 1], which checks if the next element is consecutive to the current element.
    - Increment end by 1 while the next element is consecutive.
  - Append the range string to the result list using the format str(nums[beg]) + ("->" + str(nums[end])) * (beg != end).
    - If beg is equal to end, it means there is only a single number in the range, so no arrow is appended.
    - If beg is different from end, it means there is a range of consecutive numbers, so the arrow is appended.
- Update i to end + 1 to skip the processed elements in the next iteration.
- Return the result list.

Time Complexity O(n) --> he time complexity of the provided code is O(n), where n is the length of the input array nums.
Space Complexity O(1) --> The space complexity of the code is O(1), which is constant.

In [ ]:
```
def summaryRanges(self, nums):
    i, result, N = 0, [], len(nums)

    while i < N:
        beg = end = i
        while end < N - 1 and nums[end] + 1 == nums[end + 1]: end += 1
        result.append(str(nums[beg]) + ("->" + str(nums[end])) *(beg != end))
        i = end + 1
    return result
```



```python
class Solution:
    def summaryRanges(self, nums):
        i, result, N = 0, [], len(nums)

        while i < N:
            beg = end = i
            while end < N - 1 and nums[end] + 1 == nums[end + 1]: end += 1
            result.append(str(nums[beg]) + ("->" + str(nums[end])) *(beg != end))
            i = end + 1

        return result
```

Accepted Runtime: 69 ms

Case 1 Case 2

Input

nums =

```
[0,1,2,4,5,7]
```

Output

```
["0->2","4->5","7"]
```

Expected

```
["0->2","4->5","7"]
```

Question 8

Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

Example 1:

Input: intervals = [[0,30],[5,10],[15,20]]
Output: false

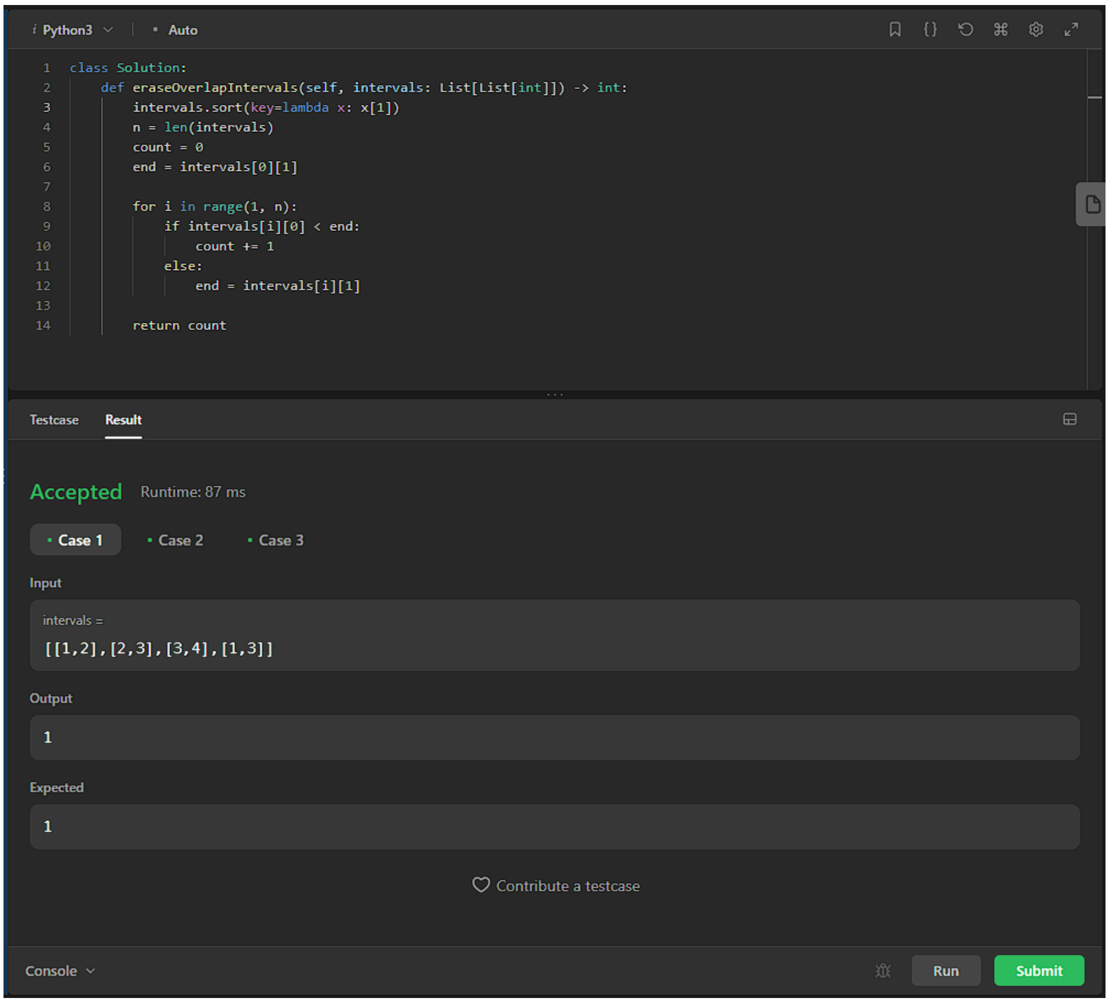
`Approach` :

- Sort the intervals based on their end time in ascending order.
- Initialize a variable count to 0 to keep track of the number of intervals to remove.
- Initialize a variable end to the end time of the first interval in the sorted array.

- Iterate through the sorted intervals starting from the second interval:
  - If the start time of the current interval is less than or equal to end, it means there is an overlap.
    - Increment count by 1 since we need to remove one of the intervals.
  - If there is no overlap, update end to the end time of the current interval.
- Return the value of count, which represents the minimum number of intervals to remove.

Time complexity O(n log n) --> Sorting the intervals takes O(n log n) time, where n is the number of intervals. The subsequent iteration through the sorted intervals takes O(n) time. Therefore, the overall time complexity is O(n log n) + O(n) = O(n log n).
Space Complexity O(n) --> Sorting the intervals requires O(n) additional space to store the sorted intervals. Therefore, the space complexity is O(n).

In [ ]:
```python
def eraseOverlapIntervals(intervals):
    intervals.sort(key=lambda x: x[1])
    n = len(intervals)
    count = 0
    end = intervals[0][1]

    for i in range(1, n):
        if intervals[i][0] < end:
            count += 1
        else:
            end = intervals[i][1]

    return count
```



```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x: x[1])
        n = len(intervals)
        count = 0
        end = intervals[0][1]

        for i in range(1, n):
            if intervals[i][0] < end:
                count += 1
            else:
                end = intervals[i][1]

        return count
```

Accepted Runtime: 87 ms

• Case 1 • Case 2 • Case 3

Input

intervals =

```
[[1,2],[2,3],[3,4],[1,3]]
```

Output

```
1
```

Expected

```
1
```