[Click on the logo to find **Problem Statement**]

Method 1: Brute Force

Intuition

The goal is to find a pair of indices in the given list (`nums`) such that the elements at those indices add up to a specified target value. The function utilizes a nested loop, with the outer loop iterating over each element in the list up to the second-to-last element, and the inner loop checking subsequent elements. By comparing the sum of pairs at each iteration with the target value, the function returns the indices of the first pair that satisfies the condition. However, the algorithm's time complexity is relatively high at O(n^2) due to the nested loop structure, making it less efficient for larger input lists. The space complexity remains constant at O(1) as the algorithm uses a fixed amount of extra space.

Approach

1. **Input Parameters:**
    - The function takes in a list of integers (`nums`) and a target integer (`target`).
2. **Loop Structure:**
    - The function uses a nested loop.
    - The outer loop iterates over each element in the list up to the second-to-last element (`length-1`).
    - The inner loop starts from the next element after the outer loop index (`i+1`) and goes up to the last element in the list.
3. **Pair Sum Comparison:**
    - Within the loops, it checks if the sum of the current pair of elements (`nums[i]` and `nums[j]`) is equal to the target.
    - If a pair is found that satisfies this condition, the function returns the indices `[i, j]` .
4. **Efficiency Consideration:**

- The algorithm has a quadratic time complexity, which may be inefficient for larger input lists.
- For better efficiency, alternative approaches such as using a hash table (dictionary) or sorting the list could be considered.

5. **Return:**
    - If a pair with the target sum is found, the function returns the indices of that pair.
    - If no such pair is found, the function does not explicitly return anything.

Complexity

- Time complexity:$O(n^2)$
- Space complexity: O(1)

Code

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        length = int(len(nums))
        for i in range(length-1):
            for j in range(i+1, length):
                if nums[i] + nums[j] == target:
                    return [i,j]
```

Method 2: Hashmap

Intuition

It employs a hashmap to store each element of the input list along with its index. The algorithm iterates through the list only once, and for each element, it calculates the difference between the target value and the current element. By checking if this difference is already present in the hashmap, the algorithm efficiently identifies a pair whose elements add up to the target. If such a pair is found, it immediately returns the corresponding indices. The use of a hashmap significantly reduces the time complexity to O(n), making the algorithm more scalable for larger input lists. The space complexity is O(n) in the worst case, as the hashmap may need to store all 'n' elements, but this is a worthwhile trade-off for the substantial improvement in time complexity compared to the previous implementation.

Approach

1. **Input Parameters:**
    - The function takes in a list of integers (`nums`) and a target integer (`target`).
2. **Hashmap Initialization:**
    - The function initializes an empty hashmap (`hashmap`) to store elements from the input list along with their indices.
3. **Iterating through the List:**
    - It iterates through the input list (`nums`) using a single loop that runs from 0 to `length-1` .
4. **Calculating Difference:**
    - For each element in the list, it calculates the difference between the target value and the current element (`difference = target - nums[i]`).
5. **Checking Hashmap:**
    - It checks if this difference is already present in the hashmap.
    - If the difference is found, it means a pair with the target sum is identified, and the function returns the indices of that pair.
6. **Updating Hashmap:**
    - If the difference is not in the hashmap, it adds the current element to the hashmap along with its index (`hashmap[nums[i]] = i`).
7. **Efficiency Consideration:**
    - This approach is more efficient than the previous nested loop approach (O(n^2)).
    - It achieves a linear time complexity by leveraging the hashmap to quickly look up elements.
8. **Return:**
    - If a pair with the target sum is found, the function immediately returns the indices of that pair.
    - If no such pair is found, the function does not explicitly return anything.

Complexity

- Time complexity: O(n)

- Space complexity: O(n)

Code

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        hashmap = {}
        length = int(len(nums))
        for i in range(length):
            diffrence = target - nums[i]
            if diffrence in hashmap:
                return [hashmap[diffrence], i]
            hashmap[nums[i]] = i
```

If you want to see more solutions to coding problems, you can visit: