**🧠 Problem: Two Sum**

**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.
* 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 <= nums.length <= 10⁴
* -10⁹ <= nums[i] <= 10⁹
* -10⁹ <= target <= 10⁹
* Only **one valid answer** exists.

**🚀 Follow-up: Can you come up with an algorithm that is less than O(n²) time complexity?**

**✅ Way 1: Brute Force (O(n²))**

**🔍 Description:**

For each element nums[i], check every other element nums[j] to its right. If nums[i] + nums[j] == target, return [i, j].

**✅ Code:**

class Solution1:  
  
 def twoSum(self, nums: List[int], target: int) -> List[int]:  
 for i in range(len(nums)):  
 for j in range(i + 1, len(nums)):  
 if nums[i] + nums[j] == target:  
 return [i, j]

**🔁 Dry Run:**

* Input: nums = [2, 7, 11, 15], target = 9
* i = 0 → nums[0] = 2  
  → j = 1 → nums[1] = 7  
  → 2 + 7 = 9 ✅ → return [0, 1]

**✅ Output:**

[0, 1]

**✅ Pros:**

* Simple and easy to understand
* No extra memory used

**❌ Cons:**

* Very slow for large input sizes (O(n²))

**✅ Way 2: Two Pointer Approach (O(n log n))**

Only applicable if you are **returning values**, not indices — or if you store indices before sorting.

**🔍 Description:**

1. Store numbers with their indices.
2. Sort based on values.
3. Use two pointers to find the pair.

**✅ Code:**

class Solution:  
 def twoSum(self, nums: List[int], target: int) -> List[int]:  
 nums\_with\_index = [(num, i) for i, num in enumerate(nums)]  
 nums\_with\_index.sort() # Sort by value  
  
 left, right = 0, len(nums) - 1  
 while left < right:  
 sum\_val = nums\_with\_index[left][0] + nums\_with\_index[right][0]  
 if sum\_val == target:  
 return [nums\_with\_index[left][1], nums\_with\_index[right][1]]  
 elif sum\_val < target:  
 left += 1  
 else:  
 right -= 1

**🔁 Dry Run:**

* Input: nums = [2, 7, 11, 15], target = 9
* After sorting with indices: [(2,0), (7,1), (11,2), (15,3)]
* Start:
  + left = 0 → 2
  + right = 3 → 15 → sum = 17 ❌
  + move right → 2 → 11 ❌
  + move right → 1 → 7 → 2 + 7 = 9 ✅ → return [0, 1]

**✅ Output:**

[0, 1]

**✅ Pros:**

* Faster than brute force
* Efficient for sorted inputs

**❌ Cons:**

* Requires sorting → extra time and space
* Must preserve original indices manually

**✅ Way 3: Using HashMap (O(n)) — Best Solution**

**🔍 Description:**

* For every element, calculate complement = target - num.
* If complement is already in the hashmap, return indices.
* Else, store the current element in the hashmap.

**✅ Code:**

from typing import List  
  
  
class Solution:  
 def twoSum(self, nums: List[int], target: int) -> List[int]:  
 num\_map = {} # To store number as key and its index as value  
  
 for i, num in enumerate(nums):  
 complement = target - num  
 if complement in num\_map:  
 return [num\_map[complement], i]  
 num\_map[num] = i  
  
  
sol = Solution()  
result = sol.twoSum([2, 7, 11, 15], 9)  
print(result) # Output: [0, 1]

**🧪 Test:**

sol = Solution()

result = sol.twoSum([2, 7, 11, 15], 9)

print(result) # Output: [0, 1]

**🧠 Step-by-Step Execution:**

1. Initialize:
2. num\_map = {}
3. **Iteration 1:**
   * i = 0, num = 2
   * complement = 9 - 2 = 7
   * 7 not in num\_map → add 2:0  
     → num\_map = {2: 0}
4. **Iteration 2:**
   * i = 1, num = 7
   * complement = 2
   * 2 is in num\_map → return [0, 1]

**✅ Final Output:**

[0, 1]

**📊 Summary Table:**

| **Approach** | **Time Complexity** | **Space Complexity** | **Notes** |
| --- | --- | --- | --- |
| Brute Force | O(n²) | O(1) | Simple but slow |
| Two Pointers | O(n log n) | O(n) | Requires sorting and tracking index |
| HashMap (Best) | O(n) | O(n) | Fastest and most efficient |