1. Two Sum

Easy Array Hash Table

Problem Description

In this problem, we have an array of integers called `nums` and a target integer called `target`. Our task is to find two distinct numbers within the array that when added together, equal the `target`. One important rule is that we cannot use the same element from the array twice in our sum. The output will be the indices (positions in the array) of these two numbers, and it does not matter in which order we provide these indices. Each valid input to the problem is guaranteed to have exactly one solution.

Intuition

To solve this problem efficiently, we avoid the brute force approach of trying every possible pair of numbers to see if they add up to the target. Instead, we employ a hash table (dictionary in Python), which offers fast lookup times that average out to O(1) complexity. The basic idea is to iterate through the list once, and for each number, we check if the number required to reach the target (`target - current_number`) has already been seen in the array. If it has, then we have found the two numbers whose sum is equal to the target.

Solution Approach

The solution provided uses a hash table to map each element's value to its index in the array. This allows for constant-time look-ups which are critical for efficiency. The algorithm proceeds as follows:

1. Initialize an empty hash table (dictionary in Python dialect), we'll call it `m`.
2. Iterate over the `nums` array, enumerating both the value `x` and its index `i`. Enumeration provides a convenient way of getting both the value and the index without additional overhead.
3. For every value `x`, calculate its complement `y` by subtracting `x` from `target` (`y = target - x`).
4. Check if `y` is present as a key in the hash table. If it is found, it means we had already seen the necessary pair earlier in the array. We then retrieve `m[y]`, which is the index of `y` we had stored, and return a list containing the indices of `y` and `x` (`[m[y], i]`). This satisfies the requirement as their sum is equal to the `target`.
5. If `y` is not in the hash table, add the current value `x` along with its index `i` to the hash table (`m[x] = i`). This stores `x` for future reference if we later come across its complement `y`.

By only traversing the array once, the overall time complexity is `O(n)`, where `n` is the number of elements in `nums`. The space complexity is also `O(n)` as in the worst case, we could potentially store all elements in the hash table before finding a match.

Here's the provided solution approach step in the code:

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        m = {}  # Step 1: Initialize the hash table
        for i, x in enumerate(nums):  # Step 2: Enumerate through nums
            y = target - x  # Step 3: Calculate complement y
            if y in m:  # Step 4: Check if y is present in the hash table
                return [m[y], i]  # Return the indices of the two elements adding up to target
            m[x] = i  # Step 5: Store x with its index in hash table for future reference
```

The strength of this approach is that it smartly utilizes the hash table to avoid nested loops and thus reducing the time complexity. The algorithm is linear time as it eliminates the need to examine every possible pair by keeping a record of what is needed to reach the target with the current number.

Example Walkthrough

Let's work through a small example to illustrate the solution approach.

Suppose our input array `nums` is `[2, 7, 11, 15]` and the `target` is `9`.

Following the solution steps:

1. Initialize an empty hash table (dictionary), which we'll name `m`.
2. Now we start iterating over the `nums` array:
   - First iteration (i = 0):
     - We take the first element `nums[0]` which is `2`.
     - Calculate its complement `y = target - nums[0]`, which gives us `y = 9 - 2 = 7`.
     - Check if `7` is present in the hash table `m`. It isn't, since `m` is empty.
     - Store the current value `2` along with its index `0` in the hash table: `m[2] = 0`.
   - Second iteration (i = 1):
     - Take the next element `nums[1]`, which is `7`.
     - Calculate its complement `y = target - nums[1]`, which gives us `y = 9 - 7 = 2`.
     - Check if `2` is in the hash table `m`. Yes, it is! The complement `2` was stored during the first iteration.
     - Since the complement is found, we retrieve `m[2]` which is `0`, the index of the complement `2`. This gives us the indices `[0, 1]`.

The sum of the numbers at these indices (`nums[0] + nums[1]`) equals the `target` (2 + 7 = 9). As expected, the original problem statement guarantees that there is exactly one solution, so the problem is now solved with the output `[0, 1]`.

By using this hash table approach, we efficiently found the two numbers that add up to the target in a single pass through the array, thereby using `O(n)` time complexity instead of `O(n^2)` which would result from using a brute force approach with nested loops.

Solution Implementation

Python

```
from typing import List  # Import List for type annotation

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        # Create a dictionary to store numbers and their indices
        index_map = {}
        # Enumerate through the list of numbers
        for index, number in enumerate(nums):
            # Calculate the complement of the current number
            complement = target - number
            # If complement is in the index_map, a solution is found
            if complement in index_map:
                # Return the indices of the two numbers
                return [index_map[complement], index]
            # Otherwise, add the current number and its index to the index_map
            index_map[number] = index
        # If no solution is found, this return will not be reached due to guaranteed solution.
```

Java

```
import java.util.Map;
import java.util.HashMap;

class Solution {
    public int[] twoSum(int[] nums, int target) {
        // Create a hashmap to store the value and its index
        Map<Integer, Integer> indexMap = new HashMap<>();

        // Iterate over the array elements
        for (int i = 0; i < nums.length; i++) {
            int current = nums[i]; // Current element value
            int complement = target - current; // The complement which, when added to 'current', equals 'target'

            // Check if the complement is already in the map
            if (indexMap.containsKey(complement)) {
                // If complement is found, return the indices of the two numbers
                return new int[] {indexMap.get(complement), i};
            }

            // Store the current value and its index in the map
            indexMap.put(current, i);
        }
        // Note: The problem statement guarantees that there will always be exactly one solution,
        // so no need to return null or throw an exception here.
        throw new IllegalArgumentException("No two sum solution found");
    }
}
```

C++

```
#include <vector>
#include <unordered_map>

class Solution {
public:
    // Function to find the indices of the two numbers that add up to a specific target.
    std::vector<int> twoSum(std::vector<int>& nums, int target) {
        // Create a hash map to store the value and its index.
        std::unordered_map<int, int> num_to_index;

        // Iterate through each number in the vector.
        for (int i = 0; i < nums.size(); ++i) {
            int current_num = nums[i];      // Current number in the iteration.
            int complement = target - current_num;  // Find the complement of the current number.

            // If the complement is found in the map, return the pair of indices.
            if (num_to_index.count(complement)) {
                return {num_to_index[complement], i};
            }

            // If complement is not found, add the current number and its index to the map.
            num_to_index[current_num] = i;
        }

        // In case no solution is found (this part is unreachable if input is guaranteed to have one solution as stated in the proble
        return {}; // Return an empty vector.
    }
};
```

TypeScript

```
// Define the function signature with input types and return type
function twoSum(nums: number[], target: number): number[] {
    // Initialize a map to store the numbers and their indices
    const numIndexMap = new Map<number, number>();

    for (let i = 0; i < nums.length; i++) {
        const currentNum = nums[i]; // The current number in the array
        const complement = target - currentNum; // The number that complements the current number to reach the target

        // Check if the complement is already in the map
        if (numIndexMap.has(complement)) {
            // If complement is found, return its index along with the current index
            return [numIndexMap.get(complement)!, i];
        }

        // If complement is not found, add current number and its index to the map
        numIndexMap.set(currentNum, i);
    }

    // Since the problem guarantees exactly one solution, the loop should never finish without returning.
    // If no solution is found (which violates the problem's constraints), throw an error.
    throw new Error('No two sum solution exists');
}

// The function `twoSum` can now be used as per its signature with TypeScript type checking.
```

```
from typing import List  # Import List for type annotation

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        # Create a dictionary to store numbers and their indices
        index_map = {}
        # Enumerate through the list of numbers
        for index, number in enumerate(nums):
            # Calculate the complement of the current number
            complement = target - number
            # If complement is in the index_map, a solution is found
            if complement in index_map:
                # Return the indices of the two numbers
                return [index_map[complement], index]
            # Otherwise, add the current number and its index to the index_map
            index_map[number] = index
        # If no solution is found, this return will not be reached due to guaranteed solution.
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the array `nums`. This is because the code loops through each element in `nums` exactly once, and each operation within the loop—including checking if an element exists in the map `m` and adding an element to `m`—is `O(1)`.

The space complexity of the code is also `O(n)`, since in the worst case, the code may insert each element of the array `nums` into the map `m`. Therefore, the space used by the map `m` grows linearly with the number of elements in `nums`.