

TwoSum Dry Run



```
import java.util.HashMap;

public class TwoSum {
    public int[] twoSum(int[] nums, int target) {
        // Create a HashMap to store the indices of elements
        HashMap<Integer, Integer> map = new HashMap<>();

        // Traverse the array
        for (int i = 0; i < nums.length; i++) {
            // Calculate the complement that would sum up to the target
            int complement = target - nums[i];

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

            // Otherwise, add the current number and its index to the map
            map.put(nums[i], i);
        }

        // Return an empty array if no solution is found (this won't happen as per the assumption)
        return new int[0];
    }

    public static void main(String[] args) {
        TwoSum solution = new TwoSum();

        // Example 1
        int[] nums1 = {2, 7, 11, 15};
        int target1 = 9;
        int[] result1 = solution.twoSum(nums1, target1);
        System.out.println("Output: [" + result1[0] + ", " + result1[1] + "]");

        // Example 2
        int[] nums2 = {3, 2, 4};
        int target2 = 6;
        int[] result2 = solution.twoSum(nums2, target2);
        System.out.println("Output: [" + result2[0] + ", " + result2[1] + "]");

        // Example 3
        int[] nums3 = {3, 3};
        int target3 = 6;
    }
}
```

```
int[] result3 = solution.twoSum(nums3, target3);
System.out.println("Output: [" + result3[0] + ", " + result3[1] + "]");
}
```

}. Dry run this for: Input: nums = [2,7,11,15], target = 9



Let's dry run the given code step-by-step for the input:

Input:

- `nums = [2, 7, 11, 15]`
- `target = 9`

Initialization:

- `map = {}` (an empty HashMap).

Step-by-Step Iteration:

Iteration 1 (`i = 0`):

- `nums[i] = 2`
- `complement = target - nums[i] = 9 - 2 = 7`
- Check if `complement (7)` exists in `map`:
 - **No**, it does not exist.
- Add `nums[i] (2)` and its index `i (0)` to `map`:
 - `map = {2: 0}`

Iteration 2 (`i = 1`):

- `nums[i] = 7`
- `complement = target - nums[i] = 9 - 7 = 2`
- Check if `complement (2)` exists in `map`:
 - **Yes**, it exists. The index of `2` in `map` is `0`.
- Return the indices of `complement` and the current number:
 - `return [map.get(complement), i] = [0, 1]`

Result:

The function returns `[0, 1]`, indicating that the numbers at indices `0` and `1` (i.e., `2` and `7`) sum up to the target `9`.

Final Output:

When the code runs with `nums = [2, 7, 11, 15]` and `target = 9`, the printed output will be:

```
makefile
```

```
Output: [0, 1]
```



So it retrurns only the first pair of indices that add upto target right



Yes, the implementation returns only the **first pair of indices** that add up to the target. As soon as the code finds a pair of numbers that satisfy the condition, it exits the loop and returns the result. This is because of the `return` statement inside the `if (map.containsKey(complement))` block.

Why only the first pair?

The algorithm is designed to:

1. Iterate through the array just once.
2. Use a `HashMap` to efficiently check for the complement.
3. Stop execution and return the result as soon as a valid pair is found.

Thus, the function does not explore any additional pairs after finding the first valid solution.

Example to Confirm:

For an input like:

```
java

int[] nums = {1, 3, 3, 4};
int target = 6;
```

The output will be `[1, 2]` because the algorithm stops at the **first valid pair** `(nums[1] + nums[2] = 3 + 3)`.

Even though `(nums[2] + nums[3] = 3 + 4)` also satisfies the condition, the function will not consider it.

If you want to find **all pairs** that add up to the target, the code would need to be modified to collect results instead of returning immediately. Let me know if you'd like help with that!

Is this conversation helpful so far?