Problem Name: Two Sum

Link: https://leetcode.com/problems/two-sum/description/

Solution Language(s): Java

Initial Approach

For this problem, I had two solutions. The first was a brute force attempt and the second is a refined solution. To begin, I chose the most simple and obvious solution — to have an outer loop through the array, *nums*, and for each value, have an inner loop through the array *nums*, starting from the current index of the outer loop. Inside of these loops would be a check to see if the outer loop's value + the inner loop's value was equal to the target value. These two indices for the outer and inner loop would then be added to an array that was returned.

Initial Solution

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if ((nums[i] + nums[j]) == target) {
                    return new int[] {i, j};
                }
            }
        }
        return null;
    }
}
```

Time Complexity: $O(n^2)$

For this solution, the time complexity suffers as there are two loops each involving the length of the array *nums* and thus an average complexity of $O(n^2)$.

Final Approach

After thinking about the problem for a couple of minutes, I realized that a HashMap could be utilized effectively here due to its O(1) complexity for .containsKey() calls. I thought about how to implement this solution and came up with the following idea: loop through the array, *nums*, and for each element in the array, calculate "target - current element" as the complementary value, and check if that Key exists in the HashMap. If it does, return an array with the Value pairing (index of the complement in the array *nums*) to the complement, along with the current index of the loop. If not, add the Key (the element) Value (the index) pair to the HashMap.

Final Solution

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] {map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return null;
    }
}
```

Time Complexity: O(n)

In this solution, the time complexity is a much faster O(n). This is because there is one loop that iterates through the array *nums* along with HashMap checks and puts which are each O(1) which yields an average complexity of O(n).

Conclusion

In conclusion, I had two solutions to this problem. The first was a brute force algorithm that involved little knowledge outside of basic Java loops. This was the most obvious and logical solution to me, and is how I would approach the problem if I was asked to do this problem without computer science — go

through each element and check if there is a complement in the rest of the list. My second solution was what I came up with while keeping in mind optimization. I saw that the time complexity for the first method was $O(n^2)$ and knew that there were ways to get the average time complexity down. My first instinct was to use a HashMap because I know that the .put() and .containsKey() methods have time complexities of O(1). This constant time complexity allowed us to remove the inner loop from the first solution and achieve a final time complexity of O(n).