Two Sum Problem - Detailed Analysis

Problem Statement:

Given an integer array nums and an integer target, return indices of the two numbers such that they add up to target. Assume exactly one solution exists and you may not use the same element twice.

1. Brute Force Approach

```
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};
        }
    }
}
```

Analysis: This approach checks every possible pair. Time Complexity: O(n^2). Space Complexity: O(1). It guarantees correctness but becomes inefficient for large inputs.

2. Optimal Approach (HashMap - Complement Lookup Pattern)

```
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);
}
```

Analysis: Instead of checking all pairs, we store previously seen numbers in a HashMap. For each element, we compute its complement and check if it already exists. Time Complexity: O(n). Space Complexity: O(n). This works because we trade memory for speed.

3. Your Approach (Counter + Single Loop)

```
int counter = 0;
for(int i = 1; i < nums.length; i++){
    if(target - nums[counter] == nums[i] && counter != i){
        return new int[]{counter, i};
    }
    counter++;
}
```

Analysis: This method only checks adjacent pairs such as (0,1), (1,2), (2,3)... It does NOT check all unique combinations. Therefore, it fails when the valid pair is not adjacent.

4. Edge Cases Where It Breaks

- Input: [1, 5, 1, 5], target = 10 → Expected indices: (1,3) but not checked.
- Any case where valid pair elements are separated by more than one index.
- Large arrays where correct pair appears early but not adjacent.

5. Deeper Pattern Analysis

Two Sum is fundamentally a pair-generation problem. For n elements, the number of unique pairs is n(n-1)/2, which grows quadratically. A single loop without additional memory can only perform O(n) checks, so it cannot examine all possible pairs. The HashMap solution succeeds because it stores history, converting the pair search into a complement lookup problem. This represents a key algorithmic principle: time-space tradeoff.