Two Sum Problem

The two sum problem involves determining if there are two elements in an array of integers that add up to a given target value. There are two variations of this problem:

1. Determine if such a pair exists (yes/no).
2. Find the indices of the two elements that sum up to the target.

Brute Force Solution 🔍

The brute force approach involves checking every possible pair of elements in the array to see if their sum equals the target.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == j) continue;
        if (array[i] + array[j] == target) {
            // Yes or return indices i, j
        }
    }
}
```

- Time Complexity: $O(n^2)$ because for every element, you check with every other element.
- Optimization: Start checking from the next element to avoid redundant checks.

  ```
  for (int i = 0; i < n; i++) {
      for (int j = i + 1; j < n; j++) {
          if (array[i] + array[j] == target) {
              // Yes or return indices i, j
          }
      }
  }
  ```

  This optimization reduces some, but not a meaningful amount, of the work.

Better Solution Using Hashing

The better solution uses **hashing** to reduce the time complexity. The thought process is to iterate through the array once, and for each element, determine what other number is required to reach the target, and check for that number in the hash map.

- Store array elements and their indices in a **hash map**.
  - **Key**: Element
  - **Value**: Index
- For each element in the array:
  - Calculate the required number: $required = target - element$
  - Check if the required number exists in the hash map.
    - If yes, return "yes" or the indices.
    - If no, add the current element and its index to the hash map.
- If the entire array is traversed without finding a solution, return "no".

Example

Given an array `[2, 6, 5, 8, 11]` and a target of `14`:

1. Start with `2`:
   - Required: `14 - 2 = 12`
   - `12` in hash map? No.
   - Add `2` with index `0` to the hash map.
2. Move to `6`:
   - Required: `14 - 6 = 8`
   - `8` in hash map? No.
   - Add `6` with index `1` to the hash map.
3. Move to `5`:
   - Required: `14 - 5 = 9`
   - `9` in hash map? No.
   - Add `5` with index `2` to the hash map.
4. Move to `8`:
   - Required: `14 - 8 = 6`
   - `6` in hash map? Yes, at index `1`.
   - Solution found: Indices `3` (current index) and `1` (from hash map).

Code

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

pair<int, int> findTwoSum(const vector<int>& nums, int target) {
    unordered_map<int, int> numMap;
    for (int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];
        if (numMap.find(complement) != numMap.end()) {
            return make_pair(numMap[complement], i);
        }
        numMap[nums[i]] = i;
    }
    return make_pair(-1, -1); // Not found
}

int main() {
    vector<int> nums = {2, 6, 5, 8, 11};
    int target = 14;
    pair<int, int> result = findTwoSum(nums, target);
    if (result.first != -1 && result.second != -1) {
        cout << "Indices: " << result.first << ", " << result.second << er
    } else {
        cout << "No solution found." << endl;
    }
    return 0;
}
```

Algorithm

~~~
map<int, int> myMap; // element -> index

for (int i = 0; i < n; i++) {
    int required = target - array[i];
    if (myMap.find(required) != myMap.end()) {
        return "yes" or indices;
    } else {
        myMap[array[i]] = i;
    }
}
return "no";
```## Two Sum

### Hashing Solution Analysis
*   The time complexity of the hashing solution is $O(n)$, assuming the ha
*   If the map works in logarithmic time, the complexity is $O(n \log n)$.
*   With an unordered map, the best and average-case time complexity is $O
*   Dumping every element into a hash map results in a $O(n)$ operation.

### Two-Pointer Approach

The **two-pointer approach** is a **greedy algorithm** used to find pairs

#### Algorithm

1.  **Sort** the input array.
2.  Initialize a **left pointer** at the beginning of the array and a **ri
3.  Calculate the **sum** of the elements at the left and right pointers.
4.  If the sum is equal to the **target**, you've found the pair.
5.  If the sum is less than the target, **increment** the left pointer to
6.  If the sum is greater than the target, **decrement** the right pointer
7.  Repeat steps 3-6 until the left and right pointers cross each other. I

#### Example

Given a sorted array `[2, 5, 6, 8, 11]` and a target of `14`:

1.  **Left pointer** starts at `2`, and **right pointer** starts at `11`.
2.  `2 + 11 = 13`, which is less than `14`. Increment the left pointer.
3.  **Left pointer** moves to `5`. Now, `5 + 11 = 16`, which is greater th
4.  **Right pointer** moves to `8`. Now, `5 + 8 = 13`, which is less than
5.  **Left pointer** moves to `6`. Now, `6 + 8 = 14`, which equals the tar

#### Code Implementation

```cpp
bool twoSum(std::vector<int>& nums, int target) {
    std::sort(nums.begin(), nums.end());
    int left = 0;
    int right = nums.size() - 1;

    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            return true;
~~~

```
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }
    return false;
}
```

Time and Space Complexity

- **Time Complexity:** $O(n \log n)$ due to sorting, plus $O(n)$ for the two-pointer traversal, resulting in $O(n \log n)$.
- **Space Complexity:** $O(1)$ if the sorting is done in place. If the space complexity of the array after sorting is taken into account, then it is $O(n)$.

Use Cases

- **Variety 1:** Determining if any two numbers in the array add up to the target value (yes/no).
- **Variety 2:** Finding the indices of the two numbers that add up to the target (less optimal).

For Variety 2, preserving the original indices requires storing the original array with indices in another data structure, sorting it, and then applying the two-pointer approach. This increases space complexity to $O(n)$. Therefore, hashing is more optimal for Variety 2.