Hashmap

Hashmap Pattern:

- Finding Pairs
- Frequency Counting
- Unique Elements
- Mapping Relationships
- Lookup or Search Operations
- Grouping or Categorization
- Detecting Patterns
- Optimizing Time Complexity
- Avoiding Nested Loops
- Storing State or Metadata

1. **Finding Pairs**: Problems that involve finding pairs of elements with a specific property, such as pairs that sum up to a target value.
2. **Frequency Counting**: Problems that require counting the frequency of elements or characters within a collection.
3. **Unique Elements**: Problems that require ensuring all elements in the collection are unique.
4. **Mapping Relationships**: Problems that involve mapping relationships between elements, such as mapping an element to its index or another related element.
5. **Lookup or Search Operations**: Problems that require efficient lookup or search operations based on the properties or values of elements.
6. **Grouping or Categorization**: Problems that involve grouping or categorizing elements based on certain criteria or properties.
7. **Detecting Patterns**: Problems that require detecting patterns or similarities between elements or subsets of elements.
8. **Optimizing Time Complexity**: Problems where using a hashmap can lead to an optimized solution with better time complexity compared to other approaches.
9. **Avoiding Nested Loops**: Problems where using a hashmap can help avoid nested loops or improve the efficiency of nested loop-based solutions.
10. **Storing State or Metadata**: Problems that require storing additional state or metadata associated with elements in the collection.

The typically involves a FOR loop and the following components

1. **Initialize Hashmap**: Create an empty hashmap (dictionary) to store elements and their indices.
2. **Iterate Through the List**: Use a loop to iterate through each element in the input list `nums`. Keep track of the current index using the `enumerate()` function.

3. **Calculate Complement**: For each element `num`, calculate its complement by subtracting it from the target value (`target - num`). This complement represents the value that, when added to `num`, will equal the target.
4. **Check if Complement Exists**: Check if the complement exists in the hashmap. If it does, it means we've found a pair of numbers that sum up to the target. Return the indices of the current element `num` and its complement from the hashmap.
5. **Store Element in Hashmap**: If the complement does not exist in the hashmap, it means we haven't encountered the required pair yet. Store the current element `num` and its index in the hashmap. This allows us to look up the complement efficiently in future iterations.
6. **Return Result**: If no such pair is found after iterating through the entire list, return an empty list, indicating that no pair of numbers sum up to the target.

```python
def find_pair_with_given_sum(nums, target):
    hashmap = {}  # Create an empty hashmap to store elements and their indices
    for i, num in enumerate(nums):
        complement = target - num  # Calculate the complement for the current element
        if complement in hashmap:
            return [hashmap[complement], i]  # If complement exists in the hashmap, return the indices
        hashmap[num] = i  # Store the current element and its index in the hashmap
    return []  # If no such pair is found, return an empty list


# Example usage:
nums = [2, 7, 11, 15]
target = 9
result = find_pair_with_given_sum(nums, target)
print("Indices of two numbers that sum up to", target, ":", result)  # Output: [0, 1] (2 + 7 = 9)
```

```python
def high(array):
    freq_map = {}
    for i in array:
        if i in freq_map:
            freq_map[i]=freq_map[i] + 1
        else:
            freq_map[i]=(1)
    return freq_map


print(high([5,7,3,7,5,6]))


{5:2 ,7:2, 3:1, 6:1}
```