# 1346. Check If N and Its Double Exist

Easy Array Hash Table Two Pointers Binary Search Sorting Leetcode Link

## Problem Description

In this problem, we are given an array `arr` consisting of integers. Our task is to determine if there are two distinct indices `i` and `j` in the array such that:

- The indices `i` and `j` are not the same (i != j).
- Both indices are within the bounds of the array (0 <= i, j < arr.length).
- The value at index `i` is double the value at index `j` (arr[i] == 2 * arr[j]).

We need to return a boolean value, true if such a pair of indices exists, and false otherwise.

## Intuition

To solve this problem, we aim to efficiently check if for any given element `arr[i]`, there exists another element `arr[j]` which is exactly half of `arr[i]`. Instead of using a brute-force approach which checks all possible combinations and results in quadratic time complexity, we can use a hash map to optimize the search process. A hash map allows for constant time lookups, reducing our overall time complexity.

The intuition behind using a hash table is:

- First, we create a hash map (denoted as `m`) that stores the value to index mapping of the array elements. This step helps us to quickly check if a value exists in the array and simultaneously ensures that the indices are distinct.
- Next, we iterate over the array, and for each element `arr[i]`, we check whether double that element (`2 * arr[i]`) exists in the hash map.
- If it does exist, we should also ensure the indices are not the same to satisfy the condition `i != j`. This is confirmed by checking if the index associated with `2 * arr[i]` in the hash map `m` is different from `i`.
- If such a condition is satisfied, we can return true, indicating that we've found two indices meeting the problem's criteria.

By hash mapping and searching for the double of the current element, we reduce the search complexity, which makes our algorithm more efficient than a naive approach.

## Solution Approach

The solution approach involves the following steps, making use of hash maps and list comprehensions in Python:

1. **Creation of a Hash Map**: A hash map `m` is created to store values from the `arr` list, mapping them to their corresponding indices. We do this using a dictionary comprehension in Python:

   ```python
   m = {v: i for i, v in enumerate(arr)}
   ```

   Here, for every element `v` in `arr`, we associate its value as the key and its index `i` as the value in the hash map `m`.

2. **Iterate and Check for Double Existence**: We iterate through the array using the `enumerate` function which gives us both the index and value.

   ```python
   return any(v << 1 in m and m[v << 1] != i for i, v in enumerate(arr))
   ```

   - For each element `v`, we check if `v << 1` (which is equivalent to `2 * v`) exists in the hash map `m`. The `<<` operator is a bit manipulation technique used for efficiency and is the same as multiplying by 2.
   - If the double exists (`v << 1 in m`), we then check if the index corresponding to that doubled value is different from the index of the current value `v` (`m[v << 1] != i`). This ensures that `i != j`.
   - The `any` function is used to check if there's at least one element in the array satisfying this condition. If such an element is found, `any` will return `True`, otherwise, it will return `False`.

In summary, this approach uses a hash map to map each value to its index and then uses list comprehension combined with bit manipulation to check efficiently if there's any element in the array that has a corresponding element that is double its value, while ensuring the indices are distinct.

### Example Walkthrough

Let's consider an example array `arr = [10, 2, 5, 3]` to illustrate the solution approach described above.

1. **Creation of a Hash Map**: We initialize an empty hash map, `m`. Then we iterate over the `arr` and fill up `m` with elements from `arr` as keys and their indices as values. After this step, the hash map looks like this:

   ```python
   m = {10: 0, 2: 1, 5: 2, 3: 3}
   ```

   The keys of the hash map `m` are the values from the array, and the values of the hash map `m` are their respective indices in the array.

2. **Iterate and Check for Double Existence**: We now use a list comprehension to check each element and its double's existence in a single pass. For each element `v` in the array, we check if `2 * v` exists in `m`:

   - Checking for the first element, `10`: We see that `10 << 1` (which is `20`) is not present in `m`. So we move to the next element.
   - Checking for the second element, `2`: `2 << 1` is `4` and is similarly not present in `m`.
   - Checking for the third element, `5`: `5 << 1` is `10`, and it is present in `m` with an index of `0`. Since the current index `2` is not the same as `0`, we have found our `i` and `j` (i=2, j=0) which satisfies `arr[i] == 2 * arr[j]`.

   There is no need to check the fourth element, `3`, as we have already found a valid pair. However, if we did, we'd see that `3 << 1` (which is `6`) is not present in `m`.

According to the list comprehension, it would return `True` after finding the valid pair for the element `5`. This confirms that such a pair with the specified criteria exists in the array.

In conclusion, for our example array `arr = [10, 2, 5, 3]`, the function implementing the solution approach would return `True`, indicating that the array contains at least one pair of indices fitting the problem's requirements.

## Python Solution

```python
from typing import List

class Solution:
    def checkIfExist(self, arr: List[int]) -> bool:
        # Create a dictionary to hold values as keys and their indices as values.
        value_to_index_map = {value: index for index, value in enumerate(arr)}

        # Iterate through each value in the array.
        for index, value in enumerate(arr):
            # Check if there is a value that is double the current value
            # and ensure that it is not the same instance as the current value.
            if (value * 2 in value_to_index_map) and (value_to_index_map[value * 2] != index):
                return True  # We found a pair where one value is double the other.

        # Return False if no pairs were found where one value is double the other.
        return False

```

## Java Solution

```java
class Solution {

    // Checks if the array contains any duplicate elements where one element
    // is twice as much as another element
    public boolean checkIfExist(int[] arr) {

        // Create a HashMap to store the array elements with their corresponding index.
        Map<Integer, Integer> elementToIndexMap = new HashMap<>();

        // Get the length of the array to iterate
        int length = arr.length;

        // Populate the HashMap with array elements as keys and their indices as values
        for (int index = 0; index < length; ++index) {
            elementToIndexMap.put(arr[index], index);
        }

        // Iterate over the array to find a duplicate where one number is double the other
        for (int index = 0; index < length; ++index) {
            // Calculate the double of the current element
            int doubleValue = arr[index] * 2;

            // Check if the double of this number exists in the map and it's not the number itself
            if (elementToIndexMap.containsKey(doubleValue) && elementToIndexMap.get(doubleValue) != index) {
                return true; // Found the pair where one is double the other
            }
        }

        // If no such pair exists, return false
        return false;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Function to check if there exists any index i such that arr[i] is equal to 2 * arr[j], where i and j are different.
    bool checkIfExist(vector<int>& arr) {
        unordered_map<int, int> indexMap; // Create a hash map to store the value to index mapping.
        int size = arr.size(); // Get the size of the array.

        // Fill the hash map with elements from the array as keys and their indices as values.
        for (int i = 0; i < size; ++i) {
            indexMap[arr[i]] = i;
        }

        // Iterate through the array to check if there's an element such that it's double exists and is not at the same index.
        for (int i = 0; i < size; ++i) {
            // Check if there is an element in the array which is double the current element and not at the same index.
            if (indexMap.count(arr[i] * 2) && indexMap[arr[i] * 2] != i) {
                return true; // If such an element is found, return true.
            }
        }

        return false; // If no such element is found, return false.
    }
};

```

## Typescript Solution

```typescript
// Function to check if the array contains any element such that
// an element is double another element in the array.
function checkIfExist(arr: number[]): boolean {
    // Create a Set to keep track of the elements we've seen so far.
    const seenElements = new Set<number>();

    // Iterate over each value in the array.
    for (const value of arr) {
        // Check if we've seen an element that is double the current value
        // or if we've seen an element that is half the current value.
        if (seenElements.has(value * 2) || (value % 2 === 0 && seenElements.has(value / 2))) {
            // If such an element exists, return true.
            return true;
        }
        // Add the current value to the set of seen elements.
        seenElements.add(value);
    }

    // If no such pair is found, return false.
    return false;
}

```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is `O(N)`, where `N` is the length of the array `arr`. This is because the code consists of two operations that scale linearly with the size of the input:

1. Creating a dictionary `m` with array values as keys and their indices as values involves a single pass through all elements in the array.
2. Then, an `any` function is used with a generator expression which may traverse up to all elements again to check the condition for each element.

However, the condition `v << 1 in m` is checked against the dictionary's keys which is an `O(1)` operation on average due to the hash table implementation of Python dictionaries. Therefore, despite the double pass, each individual check is constant time, resulting in an overall linear time complexity.

### Space Complexity

The space complexity of the code is also `O(N)`, as a dictionary `m` of size proportional to the input array is constructed. This dictionary stores each element and its respective index from the `arr`, and thus requires space that grows linearly with the number of elements in the array.