# 2766. Relocate Marbles

Medium | Array | Hash Table | Sorting | Simulation

Leetcode Link

## Problem Description

You have an array of integers, `nums`, which represents the initial positions of some marbles, indexed from `0`. Additionally, you have two more integer arrays of the same length called `moveFrom` and `moveTo`, also `0-indexed`. These arrays represent a sequence of steps you'll take to move the marbles from one position to another, respectively. For every `i` in the range of `moveFrom`, you will move all marbles located at the position given by `moveFrom[i]` to the position `moveTo[i]`. Your goal is to determine which positions are occupied by at least one marble after completing all the move steps. The final output should be a sorted list of these occupied positions.

It's important to note a couple of aspects of the problem:

- An occupied position is any position where there's at least one marble.
- Positions can be occupied by more than one marble, which means they don't necessarily decrease or increase in count after a move, as moves can be to the same position from different starting points.

## Intuition

To solve this problem, the intuition is to track the positions of marbles as moves are applied. The challenge is to do this efficiently – adjusting the positions step-by-step could become costly if we had to move each marble individually or maintain a list of marble counts at each position.

The key insight is realizing that to determine the occupied positions, we don't need to track the number of marbles at each position, just whether a position is occupied or not. A `set` data structure is perfect for this task because it can hold unique values and match the behavior we want: when marbles move from a position, we simply remove the initial position from the set, and we add the new position to the set.

A set is also useful because it automatically handles cases where multiple moves involve the same positions, as duplicate positions in a set are not possible. This means if we add an already existing moveTo position to the set, the set remains unchanged, correctly reflecting the nature of the problem.

The final step is to return a sorted list of the `set` elements, as we want the occupied positions in ascending order.

Here's the intuition behind each step of the provided solution:

1. Initialize a `set` with the starting positions from `nums`.
2. Iterate through each `moveFrom` and `moveTo` pair.
3. For each pair, remove the `moveFrom` position and add the `moveTo` position to the set.
4. Since the set only contains unique elements, it'll accurately represent all distinct occupied positions after all moves.
5. Convert the set to a sorted list to get the final positions in the required order.

## Solution Approach

The solution approach involves using a set data structure and simple for-loop iteration over the arrays that command marble moves. The choice of a set is due to its inherent properties where it stores unique elements and allows for efficient insertion and removal operations.

Let's break down the key parts of the implementation:

- **Initialization of Position Set**: The initial positions of the marbles (`nums`) are inserted into a set named `pos`. This is to ensure we have unique positions noted.

```
pos = set(nums)
```

- **Processing Moves**: A for-loop iterates over the `zip(moveFrom, moveTo)` to get pairs of start and end positions for each move. For each pair:
  - We remove the starting position `f` from the set `pos`, as that position is no longer occupied by any marble after the move:
    ```
    pos.remove(f)
    ```
  - Then, we add the new position `t` to `pos` to indicate that position is now occupied by at least one marble. If it's already occupied (i.e., already in the set), the set doesn't change, which is in line with the problem's constraints.
    ```
    pos.add(t)
    ```

This continues for all moves, iteratively updating the positions of the marbles.

- **Returning Sorted Positions**: Finally, after all moves have been applied, the set `pos` holds all the unique positions currently occupied by the marbles. We convert the set to a list and sort it to get the final answer in the required ascending order:
  ```
  return sorted(pos)
  ```

No explicit hash table is used here, but the set is internally implemented as a hash table which allows the `remove` and `add` operations to be performed in constant average time complexity, `O(1)`. The sorting at the end is `O( n log n )`, where `n` is the number of occupied positions.

Hence, the solution combines the efficiency of hash tables for updating marble positions and the necessity of returning a sorted list of the final positions. This makes the approach both intuitive and optimal for the problem at hand.

### Example Walkthrough

Let's use a small example to illustrate the solution approach:

Suppose we have:

- `nums` array as `[1, 3, 5]` representing the initial positions of the marbles.
- `moveFrom` array as `[5, 1]` representing positions from where marbles are moved.
- `moveTo` array as `[3, 4]` representing positions to where marbles are moved.

Following the solution steps:

- We first initialize a set with the initial marble positions, `pos`, which will look like this: `{1, 3, 5}`.
- Next, we need to process the moves:
  - For the first move, we get `moveFrom[0]` as `5` and `moveTo[0]` as `3`. We remove `5` from `pos` and try to add `3`. After this step, `pos` looks like this: `{1, 3}` because `3` is already present.
  - For the second move, we get `moveFrom[1]` as `1` and `moveTo[1]` as `4`. We remove `1` from `pos` and add `4` to `pos`. Now, `pos` looks like this: `{3, 4}`.
- Finally, we convert `pos` to a sorted list to get the final answer. Sorting `{3, 4}` will simply give us `[3, 4]` as the sorted list of occupied positions after all moves are completed.

So, the output for this example will be `[3, 4]` indicating these final positions are occupied by at least one marble.

## Python Solution

```python
from typing import List

class Solution:
    def relocateMarbles(self, nums: List[int], move_from: List[int], move_to: List[int]) -> List[int]:
        # Create a set from the list of initial positions to ensure uniqueness
        positions = set(nums)

        # Iterate over the pairs of move_from and move_to locations
        for source, target in zip(move_from, move_to):
            # Remove the marble's source position from the set
            positions.remove(source)
            # Add the marble's new target position to the set
            positions.add(target)

        # Return a sorted list of the final positions of marbles
        return sorted(positions)

```

## Java Solution

```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

class Solution {

    /**
     * Relocates marbles by removing the positions to move from and adding the positions to move to.
     * After all moves have been performed, the remaining positions are returned in sorted order.
     *
     * @param nums      The initial positions of the marbles.
     * @param moveFrom  Array representing the starting positions to move from.
     * @param moveTo    Array representing the ending positions to move to.
     * @return List of remaining marble positions sorted in ascending order.
     */
    public List<Integer> relocateMarbles(int[] nums, int[] moveFrom, int[] moveTo) {
        // Create a set to store the unique positions of the marbles
        Set<Integer> positions = new HashSet<>();

        // Add all initial positions to the set
        for (int num : nums) {
            positions.add(num);
        }

        // Process each move by removing the start position and adding the end position
        for (int i = 0; i < moveFrom.length; ++i) {
            positions.remove(moveFrom[i]);
            positions.add(moveTo[i]);
        }

        // Convert the set to a list to be able to sort it
        List<Integer> result = new ArrayList<>(positions);

        // Sort the list in ascending order
        result.sort((a, b) -> a - b);

        return result;
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <unordered_set>
#include <algorithm>

class Solution {
public:
    // Relocate marbles from initial positions according to move instructions.
    // Parameters:
    // nums - vector of initial marble positions
    // moveFrom - positions from which marbles are moved.
    // moveTo - positions to which marbles are moved.
    // Return:
    // returns a sorted vector of marble positions after all moves.
    vector<int> relocateMarbles(vector<int>& nums, vector<int>& moveFrom, vector<int>& moveTo) {
        // Create a set of marble positions for efficient removal and insertion.
        unordered_set<int> positions(nums.begin(), nums.end());

        // Process each move.
        for (int i = 0; i < moveFrom.size(); ++i) {
            // Remove the marble from its current position.
            positions.erase(moveFrom[i]);
            // Insert the marble into its new position.
            positions.insert(moveTo[i]);
        }

        // Convert the set back to a sorted vector to get the final marble positions.
        vector<int> finalPositions(positions.begin(), positions.end());
        sort(finalPositions.begin(), finalPositions.end());

        // Return the final positions.
        return finalPositions;
    }
};

```

## Typescript Solution

```typescript
// This function takes an array of numbers representing marble positions,
// and two arrays representing the positions to move marbles from and to.
// It will relocate each marble according to the moves specified,
// and return a sorted array of the updated marble positions.
function relocateMarbles(nums: number[], moveFrom: number[], moveTo: number[]): number[] {
    // Initialize a Set to keep track of unique marble positions.
    const positions: Set<number> = new Set();

    // Add all initial marble positions to the Set.
    nums.forEach(num => positions.add(num));

    // Iterate over the moveFrom and moveTo arrays to update positions.
    for (let i = 0; i < moveFrom.length; i++) {
        // Remove the current position from where the marble is moved.
        positions.delete(moveFrom[i]);
        // Add the new position to where the marble is relocated.
        positions.add(moveTo[i]);
    }

    // Convert the Set back to an array for sorting.
    const updatedPositions: number[] = [...positions];

    // Sort the array in ascending order.
    updatedPositions.sort((a, b) => a - b);

    // Return the sorted array of updated marble positions.
    return updatedPositions;
}

```

## Time and Space Complexity

The time complexity of the provided code can be broken down as follows:

- Creating the `pos` set from `nums` takes `O(n)` time, where `n` is the length of the `nums`.
- The zip operation does not add to the complexity since it's only creating an iterator.
- The for loop runs for the length of `moveFrom` and `moveTo` lists. In the worst case, this will be `O(n)` – assuming that each movement is unique.
- Within the loop, the `remove` operation on a set takes `O(1)` time on average, and the `add` operation also takes `O(1)` time.
- Finally, sorting the resulting set `pos` will take `O(n * log n)` time since it can have at most `n` elements from `nums`.

So, the overall time complexity is dominated by the sorting operation, which results in `O(n * log n)`.

The space complexity of the code is a result of the following:

- The set `pos`, which in the worst-case will hold all unique elements of `nums`, hence has a space complexity of `O(n)`.
- The sorted list returned is also `O(n)` space complexity since it holds the same number of elements as in the set `pos`.

Therefore, the overall space complexity is `O(n)` as it is dominated by the space required to store the unique elements from `nums`.