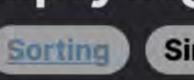
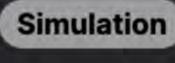
# 2154. Keep Multiplying Found Values by Two

Easy









Leetcode Link

## **Problem Description**

In this problem, you start with an integer called original and an array of integers called nums. Your goal is to perform a series of steps in which you repeatedly check whether original is in the array nums. If it is, you double the value of original and then search for the new value in the array. You continue this process of doubling and searching until original is no longer found in nums. The task is to return the final value of original after you've either doubled it several times or stopped as soon as it's not found in the array.

This problem involves the use of an iterative process to repeatedly update a number based on the contents of an array. It can be seen as a game or a search operation that has the potential to alter the target number multiple times.

### Intuition

The solution to this problem relies on efficiency and simplicity. Since we are searching for original in nums multiple times, a key insight is to use a data structure with fast lookup times. A set data structure provides O(1) average time complexity for lookups, which is ideal.

To utilize this, we first convert the list nums into a set s. Sets do not contain duplicate elements and allow us to check if an element is present in constant time.

The process is as follows:

- 1. Check if original is in the set s which contains our nums.
- 2. If it is, multiply original by two. This is efficiently done using the left shift operator <<, which basically doubles the number.
- 3. If it's not found, return the current value of original. 4. Repeat this process until original is not in s.
- The while loop in the solution keeps this process going, systematically doubling original and checking for its presence in the set s.

This loop will eventually terminate when original is no longer present in the set, at which point the most recent value of original is returned as the final result.

## The solution provided is both elegant and efficient, utilizing a set data structure and a simple while loop. Below is a step-by-step

Solution Approach

walkthrough of the implementation and the reasoning behind each step: 1. Convert the list of numbers nums to a set s:

- The conversion is done by initializing the set with nums, i.e., s = set(nums).
  - This step is crucial because it optimizes our search operation. While the lookup time for an element in a list is O(n), for a set,
  - it's O(1) on average. Despite the conversion costing O(n) time where n is the number of elements in nums, this cost is justified since we only incur
- it once, and it significantly speeds up the numerous lookups that follow. 2. Use a while loop to determine the final value of original:
- The condition for the while loop is original in s, meaning that as long as original is found in the set, the loop continues.
  - Inside the loop, original is doubled using the left shift operator, original <<= 1.</p>
  - The left shift operator effectively multiplies the number by two. Using <= 1 on an integer is equivalent to multiplying it</p> by 2.
- This is a bit manipulation trick that performs the operation quickly and with less code. 3. Return the final value of original when it's no longer found in the set.
- The algorithm's time complexity primarily depends on the number of times original can be doubled before it exceeds the largest
- number in nums. If we denote this number of doublings as k, then the overall time complexity is O(n + k), where n is the length of

nums. The space complexity is O(n) due to the additional set. To understand the practical bounds of k, consider that integers have a fixed upper limit (for example, 2^31 - 1 for 32-bit integers). The value of original will reach this upper limit after a finite number of doublings regardless of the contents of nums. Therefore, k is

bounded and in practice is much smaller than the maximum possible value of original. Thus the doubling operation does not have a significant impact on the time complexity relative to the size of nums. Example Walkthrough

# Let's walk through a small example to illustrate the solution approach.

checking if the new value is in nums, continuing this process until the value is not found.

1. Convert nums to a set s. The set s will be {1, 3, 4, 2, 8, 16}.

Suppose we have the integer original = 2 and the array nums = [1, 3, 4, 2, 8, 16]. We are tasked with doubling original and

First iteration:

2. We start a while loop that will run as long as original is in s.

- Check if original (which is 2) is in s. It is, so we proceed. We double original using the left shift operator: original <<= 1.</li>
  - Now original becomes 4.
  - Second iteration:
    - Check if the new original (which is 4) is in s. It is, so we repeat the doubling process.
  - Double original again: original is now 8. Third iteration:
  - Check if original (now 8) is in s. It is, therefore we double it again.
  - original after the left shift is now 16. Fourth iteration:
    - Doubling original gives us 32.
  - Fifth iteration: Check if original (now 32) is in s. It is not, so the while loop exits.
- 3. The value of original is now 32, and since it's not in the set s, we return 32 as the final value. Throughout this process, we have used the set for efficient lookup and doubled original easily with the left shift operator. The final

# Convert the list of numbers into a set for faster lookup

# Keep doubling the original value as long as it's found in the set

// Add all elements from the array into the set for quicker searches

// Keep doubling the 'original' value as long as it is present in the set

// Return the final value of 'original' after it can no longer be doubled

original <<= 1; // This is equivalent to multiplying 'original' by 2

result of this example would be 32, as that's the value of original when it's no longer present in nums.

Check if original (now 16) is in s. It is, so we double it once more.

class Solution: def findFinalValue(self, nums: List[int], original: int) -> int:

### original \*= 2 # Equivalent to original <<= 1 but clearer # Once the value is not found in the set, return it 10 return original 11

12

15

16

12

13

14

15

16

17

18

20

22

Python Solution

num\_set = set(nums)

while original in num\_set:

for (int num : nums) {

for (int num : nums) {

return original;

elementsSet.insert(num);

while (elementsSet.count(original) > 0) {

numSet.add(num);

```
Java Solution
   class Solution {
       /**
        * Finds the final value by doubling the original number until it's not found in the set.
        * @param nums An array of integers.
        * @param original The integer whose final value is to be found.
        * @return The final value of the original integer after doubling.
9
       public int findFinalValue(int[] nums, int original) {
10
           // Create a set to store unique elements from the array
11
           Set<Integer> numSet = new HashSet<>();
12
```

```
17
18
           // Keep doubling the original value until it's no longer found in the set
19
20
           while (numSet.contains(original)) {
               original *= 2; // equivalent to original <<= 1; for doubling
21
22
23
24
           // Return the final value of original after it couldn't be doubled any further (not found in the set)
25
           return original;
26
27 }
28
C++ Solution
   #include <unordered_set>
   #include <vector>
   class Solution {
   public:
       // Function to find the final value after doubling the original value if it is in the nums array
       int findFinalValue(std::vector<int>& nums, int original) {
           // Create an unordered_set to store the unique elements in 'nums'
           std::unordered_set<int> elementsSet;
10
           // Insert all the numbers in the 'nums' vector into the set
11
```

### 23 24 }; 25

```
Typescript Solution
 1 // Finds the final value by multiplying the original value by two as long
  // as that new value exists in the set generated from the nums array.
   function findFinalValue(nums: number[], original: number): number {
       // Initialize a new set from the nums array to facilitate O(1) lookups.
       let numberSet: Set<number> = new Set(nums);
       // Continue doubling 'original' as long as it exists within 'numberSet'.
       while (numberSet.has(original)) {
           original *= 2;
       // Return the final value, which is not present in 'numberSet'.
       return original;
```

10 11 12 13 14 } 15 Time and Space Complexity

**Time Complexity** The time complexity of the code is O(n) where n is the length of the nums list. The reasoning behind this is that the conversion of the list to a set, s = set(nums), takes linear time relative to the number of elements in the list. Then, the while loop runs at most until the value of original becomes larger than the largest element in s. In the worst-case scenario, this value could double at most n times before exceeding the max element in nums if the array contained a sequence of powers of 2. However, since the while loop checks membership in a set which is done in constant time, 0(1), the increase in original does not significantly affect the overall time complexity.

array.

**Space Complexity** 

The space complexity of the code is O(n). The extra space is used to create the set s from the list nums. The set will contain at most

n unique values where n is the number of elements in nums. Thus, the space complexity depends linearly on the size of the input