2441. Largest Positive Integer That Exists With Its Negative

Easy | Array | Hash Table | Two Pointers | Sorting

Leetcode Link

Problem Description

The problem presents us with an array of integers `nums`, which is guaranteed not to contain any zeros. Our goal is to find the largest positive integer `k` such that `-k` (the negative counterpart of `k`) is also present in the array. If such an integer `k` exists, we should return it; otherwise, we return `-1` to indicate that no such integer exists in the array.

For example, if the `nums` array is `[3, -1, -3, 4, -4]`, the largest positive integer `k` for which `-k` is also in the array is `4`. This is because both `4` and `-4` are present in the array, and there is no larger positive integer that meets the condition.

Intuition

To arrive at the solution for this problem, one may think of checking each positive integer in the array to see if its negative counterpart is also present. However, a direct approach like this might lead to excessive comparisons and result in a less efficient solution.

Instead, we can leverage a data structure that provides efficient lookups to check for the existence of elements: a set. The solution uses a set to store all the numbers from the array, which allows us to query the existence of an element in constant time.

Here is the intuitive approach of the provided solution:

1. Convert the list `nums` into a set `s` to allow for fast lookups.
2. Iterate through the set `s` and look for positive integers for which their negative counterparts also exist in `s`.
3. Use a generator expression `(x for x in s if -x in s)` to generate a sequence of such numbers on the fly.
4. Apply the `max` function to this sequence to find the largest such integer.
5. If no such element exists, the `max` function returns the `default` value of `-1`.

By doing this, the solution efficiently finds the largest integer `k` that satisfies the problem's conditions with a reduced number of element comparison operations.

Solution Approach

The implementation of the solution can essentially be broken down into the following steps, which utilize a set data structure to optimize the process:

1. **Creating a Set from the Array:** The first step in the code is to convert the list of integers `nums` into a set `s`. This conversion is crucial because it changes the time complexity of lookups from `O(n)` in a list to `O(1)` in a set, on average. Therefore, this step drastically improves the efficiency when checking for the existence of an element.

   ```
   s = set(nums)
   ```

2. **Using a Generator Expression for On-the-Fly Processing:** Instead of creating an intermediate list that contains all elements which fulfill the condition (i.e., both the number and its negative are in the set), a generator expression is used. A generator is a more memory-efficient way to handle this because it computes elements one at a time, as needed, rather than storing them all at once.

   ```
   (x for x in s if -x in s)
   ```

   In this generator expression, `x` goes through each element in set `s`. For each `x`, it checks if its negative `-x` is also a member of `s`. If both `x` and `-x` are present in `s`, `x` is considered a valid candidate for the maximum `k`.

3. **Finding the Maximum Value:** The `max` function is used here to find the largest `k` among the candidates generated by the generator expression. The `max` function is applied directly to the generator:

   ```
   max((x for x in s if -x in s), default=-1)
   ```

   The inclusion of `default=-1` ensures that if the generator expression does not yield any values (which would be the case if there is no `x` such that `-x` is also in the set), the function returns `-1`. This is the signal that there is no valid integer `k` that fits the problem's constraints.

When pieced together, these steps form an algorithm that effectively finds the largest positive `k` such that `-k` is also in the array, or returns `-1` if no such `k` exists. The usage of a set for constant-time lookups and a generator for efficient iteration is what makes this implementation both time and space-efficient.

Example Walkthrough

Let's go through a small example to illustrate the solution approach described above.

Suppose we have the following `nums` array:

```
nums = [1, 2, -2, 5, -3, -1]
```

Our objective is to find the largest positive integer `k` such that both `k` and `-k` are present in the array, or return `-1` if no such integer exists.

1. Creating a Set from the Array

We convert the list `nums` into a set `s` for efficient lookups:

```
s = set(nums)  # s will be {1, 2, -2, 5, -3, -1}
```

2. Using a Generator Expression for On-the-Fly Processing

We use a generator expression to identify positive integers in the set `s` that have their negatives present as well. We do not need to store these numbers; instead, we just generate them on the fly:

```
(x for x in s if -x in s)
```

When we iterate over `s`, we check each number:

- `1` is checked, but `-1` is in `s`. It's added to the possible candidates generated.
- `2` is checked, `-2` is also in `s`. So, `2` is a candidate.
- `-2` is skipped (since we are considering only positive `k`).
- `5` is checked, but `-5` is not in `s`. So, `5` is not a candidate.
- `-3` is skipped (since it is not positive).
- `-1` is skipped (since it is not positive).

3. Finding the Maximum Value

Now, we apply the `max` function to find the largest valid `k`. We do not need to worry about storing or sorting since the `max` function will take care of evaluating the generated sequence to find the maximum:

```
max((x for x in s if -x in s), default=-1)
```

This will output `2` because `2` and `-2` are the largest positive-negative pair in the array. If there were no such pairs, `-1` would be returned.

As a result, the solution correctly identifies `2` as the largest `k` such that `-k` is also in the array `nums`. The set and generator expression ensure the operations are done efficiently both in terms of time and space complexity.

Python Solution

```python
class Solution:
    def findMaxK(self, nums: List[int]) -> int:
        # Create a set from the input list for O(1) lookup times
        num_set = set(nums)

        # Generate a list of positive numbers for which the negative counterpart also exists in the set
        pos_nums_with_neg = (x for x in num_set if -x in num_set)

        # Find the maximum of these positive numbers. If no such number exists, return -1
        return max(pos_nums_with_neg, default=-1)

```

Java Solution

```java
class Solution {
    public int findMaxK(int[] nums) {
        // Initialize the variable to store the maximum integer k where both k and -k exist in the array
        int maxK = -1;
        // Use a HashSet to store the elements for constant time look-up
        Set<Integer> numSet = new HashSet<>();

        // Add all the elements from the input array to the HashSet
        for (int num : nums) {
            numSet.add(num);
        }

        // Iterate through the HashSet
        for (int num : numSet) {
            // Check if the negation of the current number exists in the HashSet
            if (numSet.contains(-num)) {
                // If yes, update maxK to the larger value between maxK and the current number
                maxK = Math.max(maxK, num);
            }
        }

        // Return the maximum k found, or -1 if no such k exists
        return maxK;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to find the maximum 'k' such that 'k' and '-k' both exist in the given vector.
    int findMaxK(vector<int>& nums) {
        // Create an unordered set with all elements from 'nums' to facilitate faster lookups.
        unordered_set<int> numSet(nums.begin(), nums.end());

        // Initialize the answer variable to store the maximum 'k' found.
        int maxK = -1;

        // Iterate over each number in the set.
        for (int num : numSet) {
            // Check if the negative of the current number exists in the set.
            if (numSet.count(-num)) {
                // If both 'num' and '-num' are present, update 'maxK' with the maximum so far.
                maxK = std::max(maxK, num);
            }
        }

        // Return the final answer which is the max 'k' such that both 'k' and '-k' are in 'nums'.
        return maxK;
    }
};

```

Typescript Solution

```typescript
function findMaxK(nums: number[]): number {
    // Initialize the answer to -1, which will be returned if no valid k is found.
    let maxK = -1;

    // Create a Set to store unique values from the input array for quick access.
    const numSet = new Set(nums);

    // Iterate through each number in the Set.
    for (const num of numSet) {
        // Check if the negative of the current number also exists in the Set.
        if (numSet.has(-num)) {
            // If both num and its negative are found, update maxK with the maximum value.
            maxK = Math.max(maxK, num);
        }
    }

    // Return maxK, which will be the largest k such that both k and -k exist in the array,
    // or -1 if no such k exists.
    return maxK;
}

```

Time and Space Complexity

The time complexity of the code is `O(N)`, where `N` is the length of the input list `nums`. This complexity arises because creating the set `s` requires iterating through all elements of `nums` once, and the generator expression `(x for x in s if -x in s)` iterates through the set `s` at most once. Set membership checking (i.e., `-x in s`) is `O(1)` on average due to the underlying hash table implementation, so each check is constant time.

The space complexity of the code is also `O(N)` because we are creating a set `s` with at most `N` unique values from list `nums`.