3397. Maximum Number of Distinct Elements After Operations

Why not just modify the duplicate values?

It might initially seem like the best strategy would be to simply modify the duplicate elements directly by adding or subtracting values from the range [−k,k][-k, k][−k,k] to make them distinct. However, the problem is a bit more nuanced than that. Here's why we can't just directly apply the operation to duplicates and be done with it:

Key Points to Understand:

1. **Each element can only be modified once**: You can only add or subtract once from each element in `nums`. This means that you can't modify the same element multiple times to make it distinct if it's a duplicate. You only get one shot to change each element.
2. **Maximizing distinct elements**: The goal is not just to make the duplicates distinct, but to maximize the total number of distinct elements in the entire array after applying the operations. You can't just change all duplicates to the same number (e.g., modifying every duplicate of `4` into a new number). Instead, you need to maximize the spread of distinct values across the entire array.
3. **Range of possible values**: For any element `x` in `nums`, after applying the operation, the value can be any of the values in the range [x−k,x+k][x-k, x+k][x−k,x+k]. This range includes multiple possible new values, but **there are constraints**. If you change a number to one of those new values, it might conflict with a previously changed number, which reduces the total number of distinct values.
4. **Greedy Strategy**: You have to carefully choose which value to pick for each element, ensuring that it results in a distinct number when considering

previously modified elements.

Why not just modify duplicates directly?

- Suppose you have the array `nums = [4, 4, 4, 4]` and `k = 1`.
    - If you modify the first `4` to `3`, you get `[3, 4, 4, 4]`.
    - If you modify the second `4` to `5`, you get `[3, 5, 4, 4]`.
    - The third `4` can only be modified to `5` or `3`, but both have already been used.
    - The fourth `4` also faces the same problem.

If you just try to make all the duplicates distinct by directly modifying them, you can run out of options or conflict with previous changes. This is why a more **greedy approach** is needed, where you consider the entire set of numbers and carefully choose the best way to modify them while maximizing the number of distinct elements.

Code Walk Through

```
class Solution:
    def maxDistinctElements(self, nums: List[int], k: int) -> in
        n = len(nums)

        nums.sort()

        count = 0

        # keep track of previously used value
        prev_max = -math.inf

        for i in range(n):
            # Range of value for current number
            lower_bound = nums[i] - k
            upper_bound = nums[i] + k
            if prev_max < lower_bound:
                prev_max = lower_bound
                count += 1
            elif lower_bound <= prev_max < upper_bound:
                prev_max += 1
                count += 1
            else:
                # all the value in this range is used, hence th
                continue

        return count
```

The code implements a greedy approach, leveraging the following strategy:

1. **Sorting**:
    - We first **sort the array** to make it easier to deal with duplicates and to efficiently determine the best distinct value we can create for each element.
    - Sorting ensures that when processing the elements, if a number has already been used to create a distinct value, the next element will either fit the previous value's modification or be adjusted to the next possible value.
2. **Greedy Strategy**:
    - For each element in the sorted array, we want to maximize the number of distinct values, so we consider two possible ranges for each number:
        - The number itself (within the allowed range).
        - The adjacent possible values generated by adding/subtracting any value from `[-k, k]`.
    - The idea is to select the **smallest possible distinct value** that is **greater than** the largest value used so far (tracked by `prev_max`).
3. **Tracking Distinct Values**:

- We initialize a variable `prev_max` to `-∞` (or `float('-inf')` in Python) to track the largest distinct value used so far. Initially, no values have been used, so it's set to the smallest possible value
- For each number in `nums`:
  - **If the previous distinct value (`prev_max`) is smaller than the lower bound of the current number's range (`num - k`)**, we can safely assign the **lower bound** as a new distinct value.
  - **If `prev_max` is smaller than the upper bound of the current number's range (`num + k`)**, we need to increment `prev_max` (since we can only use values greater than the previous one to maintain distinctness).

4. **Incrementing the Count**:
   - Each time we assign a new distinct value (either `num - k`, `prev_max + 1`, or `num`), we **increment the `distinct_count`**.
5. **Final Output**:
   - After processing all the numbers, the variable `distinct_count` will contain the maximum number of distinct elements that can be obtained.

Example Walkthrough:

Consider the input: `nums = [4, 4, 4, 4]`, `k = 1`.

1. **Sorting**:
   - The sorted `nums` array is `[4, 4, 4, 4]`.
2. **Step-by-Step Execution**:
   - **First Element (`4`)**:
     - `lower_bound = 4 - 1 = 3`, `upper_bound = 4 + 1 = 5`.
     - `prev_max = -∞`, so we can safely pick `3` (the lower bound) as a new distinct value.
     - `distinct_count = 1`, `prev_max = 3`.
   - **Second Element (`4`)**:
     - `lower_bound = 4 - 1 = 3`, `upper_bound = 4 + 1 = 5`.

    - `prev_max = 3`, so we need to pick a value greater than `3`. The next available value is `prev_max + 1 = 4`.
    - `distinct_count = 2`, `prev_max = 4`.
  - **Third Element (`4`)**:
    - `lower_bound = 4 - 1 = 3`, `upper_bound = 4 + 1 = 5`.
    - `prev_max = 4`, so we need to pick a value greater than `4`. The next available value is `prev_max + 1 = 5`.
    - `distinct_count = 3`, `prev_max = 5`.
  - **Fourth Element (`4`)**:
    - `lower_bound = 4 - 1 = 3`, `upper_bound = 4 + 1 = 5`.
    - `prev_max = 5`, so we need to pick a value greater than `5`. The next available value is `prev_max + 1 = 6`.
    - `distinct_count = 4`, `prev_max = 6`.

3. **Final Result**:
  - After processing all elements, the final `distinct_count` is `4`.

Time Complexity:

- **Sorting the array**: O(nlogn), where n is the length of the array.
- **Iterating through the array**: O(n), where n is the length of the array.

So, the total time complexity is O(nlogn)O(n \log n)O(nlogn), dominated by the sorting step.

Space Complexity:

- **The space complexity is O(1)**, excluding the space used by the input, since we only use a few integer variables (`distinct_count`, `prev_max`), and the array is sorted in place.

Conclusion:

- This code effectively uses a **greedy approach** to maximize the number of distinct elements, and it works efficiently for this problem.
- By sorting the array and carefully managing the largest distinct value (`prev_max`), the code ensures that we maximize the distinct values in the most efficient way possible.