719. Find K-th Smallest Pair Distance

Hard · Array · Two Pointers · Binary Search · Sorting Leetcode Link

Problem Description

The goal of this problem is to find the `k`th smallest distance between any two distinct elements within an array `nums`. The distance between a pair `(nums[i], nums[j])` is defined as the absolute difference between their values, subject to the condition that `i` is less than `j`. In other words, we're looking at all possible pairs where each element comes before the other in the array to avoid repetitions (since order doesn't matter in finding the absolute difference).

Intuition

To solve this problem, we make use of a two-part approach: sorting and binary search.

1. **Sorting**: We start by sorting the array `nums`. This step is crucial because it gives us a way to efficiently count pairs with a specific distance using a linear pass through the array. If `nums` is sorted, any pair of numbers `(a, b)` such that `b >= a+d` (where `d` is our target distance) contributes to the count of distances that are at least `d`.

2. **Binary Search**: Next, we perform a binary search, not on the original array, but on the range of possible distances (0 to the maximum difference between any two elements in `nums`). The key insight here is that if we can count how many pairs have a distance smaller than or equal to a given value, we can decide whether to move higher or lower in the range of possible distances to find our `k`th smallest distance.

3. **Count Function**: Within the binary search, we utilize a custom count function to count how many pairs have a distance less than or equal to the current distance being considered in the binary search. This count function takes advantage of the sorted nature of `nums`. For any number `b` in `nums`, we find `a` such that `a = b - current_distance` and count all numbers between `a` and `b` to get the number of pairs whose distance is at most the current distance we are checking. We use the `bisect_left` function to find the first index where `a` could be inserted to maintain the sorted order.

4. **Binary Search Application**: The `bisect_left` function is then used again, this time with our range of potential distances and passing our count function as the key. This allows us to find the smallest distance such that the number of pair distances less than or equal to it is at least `k`.

By combining sorting and binary search, the solution effectively narrows down the exact distance that is the `k`th smallest among all pairs, ensuring an efficient yet simple method to solve the problem.

Solution Approach

The solution approach largely relies on binary search and the `bisect_left` library function from Python's `bisect` module. Here's how we use these components along with some sorting:

1. **Sorting the Array**: We begin by sorting the input array `nums`. Sorting is essential because it allows us to exploit the property where for any sorted sequence, the difference between any two elements is non-decreasing as you go from left to right.

2. **Binary Search on Distances**: We perform a binary search on the range of possible distances which are from `0` to `nums[-1] - nums[0]` (the largest possible distance in the sorted array). Note that instead of the elements of the array, we're searching through potential distances, probing to find the one that will give us the `k`th smallest pair distance.

3. **The Count Function**: The `count(dist)` function is a helper function that counts how many pairs in the sorted `nums` array have a distance less than or equal to `dist`. We do this by iterating over each element `b` in the array and using `bisect_left` to find the first index where `b - dist` could fit (this would be the element `a`). The count of elements from this index to `i` (current index of `b`) gives us the number of pairs with one element less than or equal to `b - dist` and the other equal to `b`. Thereby, the difference between the indices gives us the pairs count with a maximum distance of `dist`.

4. **Using Bisect to Find the Kth Distance**: Finally, `bisect_left` is used on the range of possible distances with our `count` function as the key. This means that for each potential distance `dist` that the binary search is considering, `bisect_left` uses the `count` function to determine if the number of pairs with a distance less than or equal to `dist` is less than `k`. If the count is less than `k`, `bisect_left` discards the left part (smaller distances) and moves right (towards larger distances) to find the first distance where the number of pairs is at least `k`.

In essence, what we've done is utilize binary search in a somewhat unconventional manner. Instead of searching for a value in the array, we're searching for the smallest value within a range (here, a calculated range of distances) such that a condition (count of pairs with at least this distance) holds true, therefore finding the `k`th smallest distance pair in the array.

Example Walkthrough

To illustrate the solution approach, let's go through an example. Suppose we have the array `nums = [1, 6, 11, 14]`, and we want to find the 3rd smallest distance between any two distinct elements.

1. **Sorting the Array**: First, we sort the array. In this case, the array is already sorted: `[1, 6, 11, 14]`.

2. **Binary Search on Distances**: We identify the range of the possible distances. The smallest distance is `0` (for identical elements), and the largest is `14 - 1 = 13`.

3. **The Count Function**: Now, we need a function to count pairs with distance less than or equal to `dist`. For example, if `dist = 4`, we would find:

   - Pairs for `b = 6`: None, since no `a` exists in `[1]` such that `6 - a <= 4`.
   - Pairs for `b = 11`: The element that fits is `6` (`11 - 6 = 5` is not less than or equal to `4`), so there are 0 pairs.
   - Pairs for `b = 14`: The element that fits is `11` (`14 - 11 = 3` is less than or equal to `4`), so there is 1 pair `[11, 14]`.

   Thus for `dist = 4`, the count is 1.

4. **Using Bisect to Find the Kth Distance**: We apply a binary search (using `bisect_left`) to the range `[0, 13]` to find the smallest distance `dist` such that there are at least `k = 3` pairs with distance less than or equal to `dist`.

Now, we execute the steps of binary search to find the 3rd smallest pair distance:

- **First Iteration**: Check `mid = (0 + 13) / 2 = 6.5`, rounded down to `6`. Using the count function, we find there are three pairs with distances less than or equal to `6`: `[1, 6]`, `[6, 11]`, and `[11, 14]`. Since we have found exactly 3 pairs, we know `dist = 6` could be the 3rd smallest distance.

- **Second Iteration**: Now to ensure that `6` is the smallest possible distance with at least 3 pairs, we need to check if any smaller number of pairs can still meet the `k = 3` condition. We check left of `6`, using `mid = (0 + 6) / 2 = 3`. The pairs with distance less than or equal to `3` are `[11, 14]`, so just one pair, which is not enough.

- **Third Iteration**: Since `3` is too small, we now check between `4` and `6`. Let `mid = (4 + 6) / 2 = 5`. Now, the count function returns two pairs with distances less than or equal to `5`: `[1, 6]` and `[11, 14]`. Still not enough.

- **Fourth Iteration**: Since `5` is too small and `6` is a possible solution but might not be the smallest one, we must check now if `6` really is the smallest. We check `dist = 6` directly as there are no other integers between `5` and `6`. We reaffirm there are three pairs `[1, 6]`, `[6, 11]`, and `[11, 14]`.

Since we cannot find a smaller distance that gives us at least `k` pairs, we conclude that the 3rd smallest distance is indeed `6`.

Hence, following the approach outlined, we successfully applied a binary search on the range of distances, utilizing a sorted array and a counting function, to identify the 3rd smallest distance between two elements in the array `nums = [1, 6, 11, 14]`, which is `6`.

Python Solution

```python
from bisect import bisect_left

class Solution:
    def smallestDistancePair(self, nums: List[int], k: int) -> int:
        # Helper function to count the number of pairs with distance less than or equal to 'dist'
        def count_pairs_with_max_distance(dist):
            count = 0
            # Iterate over the sorted list of numbers
            for i, upper_bound in enumerate(nums):
                # Find lower bound such that upper_bound - lower_bound <= dist
                lower_bound = upper_bound - dist
                # Find the first position where lower_bound can be inserted to maintain sorted order
                insertion_point = bisect_left(nums, lower_bound, 0, i)
                # Accumulate the count of pairs where the distance is less than or equal to 'dist'
                count += i - insertion_point
            return count

        nums.sort()  # First, sort the numbers to make it easier to find pairs

        # The maximum possible distance is the difference between the largest and smallest numbers
        max_possible_distance = nums[-1] - nums[0]

        # Binary search for the smallest distance such that there are at least 'k' pairs with that distance or less
        # The search is conducted over the range of possible distances
        smallest_distance = bisect_left(range(max_possible_distance + 1), k, key=count_pairs_with_max_distance)

        return smallest_distance

# Note: `List[int]` should be imported from typing module if type hints are used.

```

Java Solution

```java
class Solution {

    // Function to calculate the kth smallest distance pair.
    public int smallestDistancePair(int[] nums, int k) {
        // First, sort the array to make distances easier to manage.
        Arrays.sort(nums);

        // Initialize binary search bounds.
        int left = 0;
        int right = nums[nums.length - 1] - nums[0];

        // Perform binary search to find the smallest distance.
        while (left < right) {
            // Find the middle distance.
            int mid = (left + right) >> 1;

            // Count the number of pairs with distance less than or equal to mid.
            if (countPairsWithMaxDistance(mid, nums) >= k) {
                // If count is greater than or equal to k, narrow the search to the left half.
                right = mid;
            } else {
                // Otherwise, narrow the search to the right half.
                left = mid + 1;
            }
        }

        // The left boundary will be the result when search is narrowed down to one element.
        return left;
    }

    // Helper function to count pairs with maximum distance "dist".
    private int countPairsWithMaxDistance(int dist, int[] nums) {
        int count = 0;

        // Iterate over the array to count pairs.
        for (int i = 0; i < nums.length; ++i) {
            // Perform a binary search for the index of the first element in the array
            // that is within the desired distance from nums[i].
            int left = 0;
            int right = i;

            while (left < right) {
                // Calculate mid index.
                int mid = (left + right) >> 1;

                // Calculate the target distance.
                int target = nums[i] - dist;

                // Narrow down the search based on the distance to the target.
                if (nums[mid] >= target) {
                    right = mid;
                } else {
                    left = mid + 1;
                }
            }
            // Count the number of pairs with a difference of at most "dist" for current index "i".
            count += i - left;
        }

        return count;
    }
}

```

C++ Solution

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

class Solution {
public:
    // This function computes the k-th smallest distance pair in a list of integers.
    int smallestDistancePair(vector<int>& nums, int k) {
        // First, the array is sorted to calculate distances between pairs efficiently.
        sort(nums.begin(), nums.end());

        // Initialize binary search bounds. The smallest distance can be zero,
        // and the largest possible distance is between the smallest and largest element.
        int left = 0;
        int right = nums.back() - nums.front();

        // Perform a binary search to find the k-th smallest distance.
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to dividing by 2.

            // If the count of distances less than or equal to mid is greater than or equal to k,
            // we adjust the right bound.
            if (countPairsWithDistance(nums, mid) >= k)
                right = mid;
            else // Otherwise, we adjust the left bound.
                left = mid + 1;
        }

        // Once left meets right, we've found the smallest distance.
        return left;
    }

private:
    // Helper method to count the number of pairs with distance less than or equal to "dist".
    int countPairsWithDistance(const vector<int>& nums, int dist) {
        int count = 0;
        for (int i = 0; i < nums.size(); ++i) {
            // For each number, find the first number that is not more than "dist" away
            // from the current number by using lower_bound.
            int target = nums[i] - dist;
            int j = lower_bound(nums.begin(), nums.end(), target) - nums.begin();

            // Add the number of valid pairs found with this distance.
            count += i - j;
        }
        return count;
    }
};

```

Typescript Solution

```typescript
function smallestDistancePair(nums: number[], k: number): number {
    // Sort the input array in ascending order.
    nums.sort((a, b) => a - b);

    // Store the length of nums array.
    const n = nums.length;

    // Initialize left pointer to the smallest distance
    // and right pointer to the largest distance.
    let left = 0;
    let right = nums[n - 1] - nums[0];

    // Use binary search to find the kth smallest distance.
    while (left < right) {
        // Calculate the mid value as the average of left and right.
        let mid = left + ((right - left) >> 1);
        let count = 0; // Initialize the counter for pairs within distance.
        let start = 0; // Start pointer for the sliding window.

        // Iterate over the nums array using a sliding window approach.
        for (let end = 0; end < n; end++) {
            // Increment the start pointer to maintain a maximum distance of mid
            // between start and end pointers.
            while (nums[end] - nums[start] > mid) {
                start++;
            }
            // Count pairs with distance less than or equal to mid.
            count += end - start;
        }

        // If the number of distances less than or equal to mid is at least k,
        // narrow the range to the left half (search for a smaller distance).
        if (count >= k) {
            right = mid;
        } else {
            // Otherwise, narrow the range to the right half (search for a larger distance).
            left = mid + 1;
        }
    }

    // When left and right converge, left (or right) is the kth smallest distance.
    return left;
}

```

Time and Space Complexity

The time complexity of the given code mainly consists of two parts: the sorting of the `nums` list and the binary search which uses the `count` function at each step.

- Sorting the list `nums` involves a time complexity of `O(N log N)`, where `N` is the number of elements in `nums`.

- The binary search is performed on a range from 0 to `nums[-1] - nums[0]`, which might seem like it could have a maximum of `N` possible values (if all numbers in `nums` are distinct). There are `log(N)` iterations due to the binary search process.

- At each step of the binary search, we invoke the `count` function. The `count` function itself has a time complexity of `O(N)` since it involves iterating over `nums` to calculate the count of pairs with a distance less than or equal to `dist`.

Combining these, the total time complexity of the main binary search loop is `O(N log N)` for the binary search multiplied by the `O(N)` complexity of the `count` function, that is `O(N log N * N)`, or simply `O(N^2 log N)`.

The space complexity of the algorithm is `O(1)` if we exclude the input and the space used for sorting `nums`, which is typically `O(log N)` for the sort stack space in the case of efficient sorting algorithms like Timsort used in Python's `sorted()` function. Otherwise, including the space used for sorting, it would be `O(N)` if we consider that sorting might be done out-of-place requiring additional space proportional to the input size.