658. Find K Closest Elements

Medium Array Two Pointers Binary Search Sorting Sliding Window Heap (Priority Queue)

Problem Description

Given a sorted array of integers `arr`, and two integers `k` and `x`, the task is to find the `k` closest integers to `x` in the array. The result should be returned in ascending order. To determine which integers are the closest, we follow these rules:

1. An integer `a` is considered closer to `x` than an integer `b` if the absolute difference `|a - x|` is less than `|b - x|`.
2. If the absolute differences are the same, then `a` is closer to `x` than `b` if `a` is smaller than `b`.

In essence, the problem asks us to find a subsequence of the array that contains integers that are nearest to `x`, with a special emphasis on the absolute difference and the value of the integers themselves when the differences are the same.

Intuition

To solve the problem, we need an efficient way to find the subsequence of length `k` out of the sorted array that is closest to the target integer `x`.

Method 1: Sort A brute force approach would be to sort the elements of the array based on their distance from `x`. After sorting, we can simply take the first `k` elements. However, this method is not optimal in terms of time complexity because sorting would take O(n log n) time.

Method 2: Binary search A more efficient approach utilizes the fact that the array is already sorted. We know that the `k` closest integers we are looking for must form a consecutive subsequence in the array. Instead of looking for each integer separately, we should look for the starting index of this subsequence. We can use a binary search strategy to find this starting index. The high-level steps are:

1. Set two pointers: `left` at the start of the array and `right` at the position `len(arr) - k` because any subsequence starting beyond this point does not have enough elements to be `k` long.
2. Perform a binary search between `left` and `right`:
    - Calculate the middle index `mid`.
    - Check if `x` is closer to `arr[mid]` or to `arr[mid + k]`.
    - If `x` is closer to `arr[mid]` or at the same distance to both, we discard the elements to the right of `mid + k` because the best starting point for our subsequence must be to the left or inclusive of `mid`.
    - If `x` is closer to `arr[mid + k]`, we discard the elements to the left of `mid` since the best starting point must be to the right.
3. Keep narrowing down until `left` is equal to `right`, indicating that we have found the starting index of the `k` closest elements.

Solution Approach

The solution implements a binary search algorithm to efficiently find the starting index for the `k` closest elements to `x`. Here's a step-by-step breakdown of the approach and the code implementation:

1. We begin by initializing two pointers, `left` at `0` and `right` at `len(arr) - k`. This is because the `k` elements we seek must form consecutive entrants in the array, and starting any further to the right would leave insufficient elements to reach a count of `k`.

2. We enter a loop that continues until `left` is no longer less than `right`, indicating that we have narrowed down to the exact starting index for the `k` closest elements.

3. Inside the loop, we find the middle index between `left` and `right` using the formula `mid = (left + right) >> 1`. The `>> 1` is a bitwise right shift that effectively divides the sum by 2.

4. With `mid` established, we check the distance between `x` and the current middle value at `arr[mid]` compared to the distance between `x` and the potential upper bound for our sequence at `arr[mid + k]`.

    - If the element at `mid` is closer to `x` (or the elements are equally distant), we know that the `k` closest elements can't start any index higher than `mid`. Thus we set `right` to `mid`, moving our search space leftward.
    - Conversely, if the element at `mid + k` is closer to `x`, it means the sequence starts further to the right, so we move our `left` pointer up to `mid + 1`.

5. By repeatedly halving our search space, we eventually arrive at a point where `left` equals `right`. This index represents the start of the subarray containing the `k` closest numbers to `x`.

6. Once we have the starting index (`left`), we slice the original array from `left` to `left + k` to select the `k` closest elements.

7. This slice is then returned, and since the original array was sorted, this resulting slice is guaranteed to be sorted as well, yielding the correct answer.

Using this binary search approach, we achieve a time complexity of O(log(n - k)) for finding the subsequence, which is an optimization over a complete sort-based method that would require O(n log n) time. This optimization is significant for large datasets where the difference in time complexity could be substantial.

Example Walkthrough

Let us walk through a simple example to illustrate the solution approach.

Example

Assume we have an array `arr = [1, 3, 4, 7, 8, 9]`, and we want to find `k = 3` closest integers to `x = 5`.

Initial Setup

- We begin by initializing two pointers, `left` is set to `0` and `right` is set to `len(arr) - k`, which in this case is `6 - 3 = 3`. The subarray `[7, 8, 9]` is the last possible sequence of `k` numbers, and we will not consider starting indices beyond this.

Binary Search

- The binary search begins. We calculate the middle index between `left` and `right`. The initial `mid` is `(0 + 3) / 2 = 1.5`, rounded down to `1`.
- We then compare the distance of `x = 5` from `arr[mid] = 3` and from `arr[mid + k] = arr[4] = 8`. The distance from `3` is `|5 - 3| = 2` and from `8` is `|5 - 8| = 3`.
- Since `3` is closer to `5` than `8` is, we set `right` to `mid`, which is now `1`. Our search space for starting indices is now `[0, 1]`.

Narrowing Down

- We perform another iteration of the binary search. Our new `mid` is `(0 + 1) / 2 = 0.5`, rounded down to `0`.
- Again, we compare distances from `x = 5`: `arr[mid] = 1` has a distance of `4`, and `arr[mid + k] = arr[3] = 7` has a distance of `2`.
- Since `7` is closer to `5` than `1` is, we move the `left` pointer to `mid + 1`, which is `1`.

Convergence

- Because `left` now equals `right`, we have converged to the starting index for the `k` closest elements. Our starting index is `1`.

Obtaining the Result

- We slice the original array from `left` to `left + k`, which yields `[3, 4, 7]`.

Conclusion

From our example using the array `arr = [1, 3, 4, 7, 8, 9]` and the values `k = 3` and `x = 5`, we have walked through the binary search approach and determined that the `k` closest integers to `x` in this array are `[3, 4, 7]`, as illustrated by the binary search logic to find the starting index and the subsequent slice of the array. This method efficiently finds the correct subsequence without fully sorting the array based on proximity to `x`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        # Initialize the left and right pointers for binary search.
        # The right pointer is set to the highest starting index for the sliding window.
        left, right = 0, len(arr) - k

        # Perform binary search to find the left bound of the k closest elements.
        while left < right:
            # Calculate the middle index between left and right.
            mid = (left + right) // 2

            # Check the distance from the x to the middle element and the element at mid + k position.
            # If the element at mid is closer to x or equal in distance compared to the element at mid + k,
            # we move the right pointer to mid. Otherwise, we adjust the left pointer to mid + 1.
            if x - arr[mid] <= arr[mid + k] - x:
                right = mid
            else:
                left = mid + 1

        # Extract the subarray from the left index of size k, which will be the k closest elements.
        return arr[left:left + k]

# Example usage:
# sol = Solution()
# result = sol.findClosestElements([1, 2, 3, 4, 5], k=4, x=3)
# print(result)  # Output should be [1, 2, 3, 4]
```

Java

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

class Solution {
    public List<Integer> findClosestElements(int[] arr, int k, int x) {
        // Initialize the left and right pointers for binary search.
        int left = 0;
        int right = arr.length - k;

        // Continue searching until the search space is reduced to a single element.
        while (left < right) {
            // Calculate the middle index.
            int mid = left + (right - left) / 2; // Using left + (right - left) / 2 to avoid potential overflow.

            // If the distance to the left is less than or equal to the distance to the right,
            // we need to move towards the left (lower indices).
            if (x - arr[mid] <= arr[mid + k] - x) {
                right = mid;
            } else {
                // Otherwise, move right (higher indices).
                left = mid + 1;
            }
        }

        // Create a list to store the k closest elements.
        List<Integer> result = new ArrayList<>();
        // Add the closest elements to the list, starting from the left pointer.
        for (int i = left; i < left + k; ++i) {
            result.add(arr[i]);
        }

        // Return the list of k closest elements.
        return result;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    std::vector<int> findClosestElements(std::vector<int>& arr, int k, int x) {
        // Initialize the binary search bounds
        int left = 0;
        int right = arr.size() - k;

        // Perform binary search to find the start index of the k closest elements
        while (left < right) {
            // Calculate mid index (avoid potential overflow by using left + (right-left)/2)
            int mid = left + (right - left) / 2;

            // Compare the differences between x and elements at mid index and mid+k index
            // The goal is to find the smallest window such that the elements are closest to x
            if (x - arr[mid] <= arr[mid + k] - x) {
                // If the element at mid index is closer to x, or equally close
                // as the element at mid+k index (prefer the smaller element),
                // move the right bound to mid
                right = mid;
            } else {
                // Otherwise, if the element at mid+k index is closer to x,
                // move the left bound to mid + 1
                left = mid + 1;
            }
        }

        // Create and return a vector of k closest elements starting from the 'left' index
        return std::vector<int>(arr.begin() + left, arr.begin() + left + k);
    }
};
```

TypeScript

```typescript
/**
 * Finds the k closest elements to a given target x in a sorted array.
 *
 * @param {number[]} arr - The sorted array of numbers.
 * @param {number} k - The number of closest elements to find.
 * @param {number} x - The target number to find the closest elements to.
 * @returns {number[]} - An array of k closest elements to the target.
 */
function findClosestElements(arr: number[], k: number, x: number): number[] {
    // Initialize two pointers for binary search.
    let leftPointer = 0;
    let rightPointer = arr.length - k;

    // Binary search to find the start index of the k closest elements.
    while (leftPointer < rightPointer) {
        const midPointer = (leftPointer + rightPointer) >> 1; // Equivalent to Math.floor((left + right) / 2)

        // If the element at the middle is closer to or as close as the element k away from it,
        // then the closer elements are to the left side of the array.
        // Otherwise, they are to the right, and we move the left pointer one step past the middle.
        if (x - arr[midPointer] <= arr[midPointer + k] - x) {
            rightPointer = midPointer;
        } else {
            leftPointer = midPointer + 1;
        }
    }

    // Slice the array from the left pointer to get the k closest elements.
    return arr.slice(leftPointer, leftPointer + k);
}
```

```python
from typing import List

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        # Initialize the left and right pointers for binary search.
        # The right pointer is set to the highest starting index for the sliding window.
        left, right = 0, len(arr) - k

        # Perform binary search to find the left bound of the k closest elements.
        while left < right:
            # Calculate the middle index between left and right.
            mid = (left + right) // 2

            # Check the distance from the x to the middle element and the element at mid + k position.
            # If the element at mid is closer to x or equal in distance compared to the element at mid + k,
            # we move the right pointer to mid. Otherwise, we adjust the left pointer to mid + 1.
            if x - arr[mid] <= arr[mid + k] - x:
                right = mid
            else:
                left = mid + 1

        # Extract the subarray from the left index of size k, which will be the k closest elements.
        return arr[left:left + k]

# Example usage:
# sol = Solution()
# result = sol.findClosestElements([1, 2, 3, 4, 5], k=4, x=3)
# print(result)  # Output should be [1, 2, 3, 4]
```

Time and Space Complexity

The provided algorithm is a binary search approach for finding the `k` closest elements to `x` in the sorted array `arr`. Here's the analysis:

- **Time Complexity:** The time complexity of this algorithm is `O(log(N - k))` where `N` is the number of elements in the array. This is because the binary search is performed within a range that's reduced by `k` elements (since we're always considering `k` contiguous elements as a potential answer).

  The algorithm performs a binary search by repeatedly halving the search space, which is initially `N - k`. It does not iterate through all `N` elements, but narrows down the search to the correct starting point for the sequence of `k` closest elements. After finding the starting point, it returns the subarray in constant time, as array slicing in Python is done in `O(1)` time.

  Hence, the iterative process of the binary search dominates the running time, which leads to the `O(log(N - k))` time complexity.

- **Space Complexity:** The space complexity of this algorithm is `O(1)`. The code uses a fixed amount of extra space for variables `left`, `right`, and `mid`. The returned result does not count towards space complexity as it is part of the output.

Please note that array slicing in the return statement does not create a deep copy of the elements but instead returns a reference to the same elements in the original array, thus no extra space proportional to `k` or `N` is used in this operation.