Solution: Find K Closest Elements

Let's solve the Find K Closest Elements problem using the Modified Binary Search pattern.

We'll cover the following

Statement

You are given a sorted array of integers, `nums`, and two integers, `target` and `k`. Your task is to return `k` number of integers that are close to the target value, `target`. The integers in the output array should be in a sorted order.

An integer, `nums[i]`, is considered to be closer to `target`, as compared to `nums[j]` when |`nums[i]` - `target`| $<$ |`nums[j]` - `target`|. However, when |`nums[i]` - `target`| $=$ |`nums[j]` - `target`|, the smaller of the two values is selected.

Constraints:

- $1 \leq$ `k` $\leq$ `nums.length`
- $1 \leq$ `nums.length` $\leq 10^4$
- `nums` is sorted in ascending order.
- $-10^4 \leq$ `nums[i]`, `target` $\leq 10^4$

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The `k` closest integers to `target` are those integers of `nums` that have the minimum distance from `target`, and this distance is the absolute difference between the integer and `target`.

In the naive approach, we first compute the distance of every element of `nums` from the given `target`. We store these distances along with the elements themselves as pairs in a new array, `distances`, that is, each pair will consist of the absolute difference and the corresponding element from `nums`. Now, we sort `distances` based on the absolute differences in ascending order. However, if any two pairs have the same absolute difference, sort them based on the element value in ascending order. Next, we iterate through the sorted distances to extract and store the required k elements in a new array, result. Finally, we sort result and return it as the output.

For example, if nums = [1, 2, 3, 4], target = 3, and k = 2, then distances = [(2, 1), (1, 2), (0, 3), (1, 4)]. It will get sorted like this: [(0, 3), (1, 2), (1, 4), (2, 1)]. Now, pick the first two elements, i.e., result = [3, 2]. Sort result to get the valid output, i.e., [2, 3].

We traverse the complete array to calculate the distance of each element in nums from target, so it takes $O(n)$ time, where n is the number of elements in nums. Then, we have to sort two arrays, and the time complexity of the best sorting algorithm is $O(nlogn)$. Sorting distances takes $O(nlogn)$, and sorting result takes $O(klogk)$. Extracting and storing k elements in result takes $O(k)$. Therefore, the overall time complexity for the naive approach is $O(n + nlogn + k + klogk)$. In the worst case, when $k = n$, it simplifies to $O(nlogn)$.

The space complexity to store the absolute distances of elements of nums from target is $O(n)$. Additionally, it takes $O(k)$ to store k elements in result. Therefore, the overall space complexity is $O(n + k)$ which, in the worst case, when $k = n$ simplifies to $O(n)$.

Optimized approach using modified binary search

Before we proceed to the optimized approach, a few points need some consideration:

- If the length of nums is the same as the value of k, return all the elements.
- If target is less than or equal to the first element in nums, the first k elements in nums are the closest integers to target. For example, if nums= [1, 2, 3], target= 0, and k = 2, then the two closest integers to target are [1, 2].
- If target is greater than or equal to the last element in nums, the last k elements in nums are the closest integers to target. For example, if nums= [1, 2, 3], target= 4, and k = 2, then the two closest integers to target are [2, 3].
- Otherwise, we search for the k closest elements in the whole array.

When we have to find k elements in the complete array, instead of traversing the whole array, we can use binary search to limit our search to the relevant parts. The optimized approach can be divided into two parts:

1. Use binary search to find the index of the first closest integer to target in nums.
2. Use two pointers, windowLeft and windowRight, to maintain a sliding window. We move the pointers conditionally, either towards the left or right, to expand the window until its size gets equal to k. The k elements in the window are the k closest integers to target.

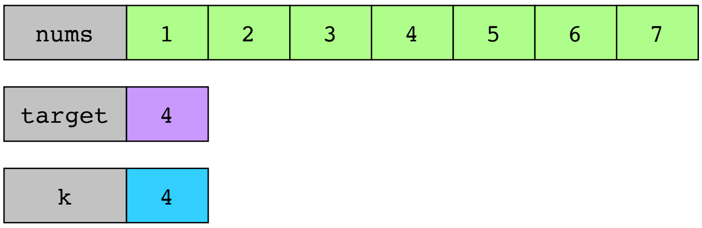
Here's how we'll implement this algorithm:

- If the length of nums is the same as k, return nums.
- If target $\leq$ nums[0], return the first k elements in nums.
- If target $\geq$ nums[nums.length - 1], return the last k elements in nums.
- Use binary search to find the index, firstClosest, of the closest integer to target.
  - Initialize two pointers, left and right, to 0 and nums.length - 1, respectively.
  - Calculate the index of the middle pointer, mid, and check:
    - If the value pointed to by mid is equal to target, i.e., nums[mid] $=$ target, return mid.
    - If nums[mid] $<$ target, move left toward the right.

        - If `nums[mid]` $>$ `target`, move `right` toward the left.
    - Once we have found the closest element to `target`, return the index, `firstClosest`, which points to it.
- Create two pointers, `windowLeft` and `windowRight`. The `windowLeft` pointer initially points to the index of the element that is to the left of `nums[firstClosest]`, and `windowRight` points to the element that is to the right of `nums[firstClosest]`. This means `windowLeft` = `nums[firstClosest] - 1`, and `windowRight` = `nums[firstClosest] + 1`.
- Traverse `nums` while the sliding window size is less than `k`. In each loop, adjust the window size by moving the pointers as follows:
    - If `nums[windowLeft]` is closer to `target` than `nums[windowRight]`, or if both are at equal distance, that is, `|nums[windowLeft] - target|` $\leq$ `|nums[windowRight] - target|`, then `windowLeft` $=$ `windowLeft - 1`.
    - If `nums[windowRight]` is closer to `target` than `nums[windowLeft]`, that is, `|nums[windowRight] - target|` $<$ `|nums[windowLeft] - target|`, then `windowRight` $=$ `windowRight + 1`.

5. Once we have `k` elements in the window, return them as the output.

The slides below help to understand the solution in a better way.



Find 4 closest elements in nums to the given target, 4.

1 of 9

Let's look at the code for this solution below:

Java

main.java

BinarySearch.java

```
class KClosest {
    public static List<Integer> findClosestElements(int[] nums, int k, int ta

        List<Integer> closestElements = new ArrayList<>();

        // If the length of 'nums' is the same as k, return 'nums'
        if (nums.length == k) {
            for (int num : nums) {
                closestElements.add(num);
            }
```

```
            return closestElements;
        }

        // if target is less than or equal to first element in 'nums',
        // return the first k elements from 'nums'
        if (target <= nums[0]) {
            for (int i = 0; i < k; i++) {
                closestElements.add(nums[i]);
            }
            return closestElements;
        }

        // if target is greater than or equal to last element in 'nums',
        // return the last k elements from 'nums'
        if (target >= nums[nums.length - 1]) {
            for (int i = nums.length - k; i < nums.length; i++) {
                closestElements.add(nums[i]);
```

Find K Closest Elements

Solution summary

To summarize, we use binary search to locate the first closest element to `target`, then create a sliding window using two pointers to select the `k` closest elements. The window adjusts its size by moving the pointers based on which adjacent element is closer to the target. Eventually, the window will have the required `k` elements, which are then returned.

Time complexity

The time complexity of the binary search is $O(logn)$, where n is the length of the input array `nums`. The sliding window step involves traversing the array once while adjusting the window size, and it takes $O(k)$ time. The overall time complexity becomes $O(logn + k)$.

Space complexity

Alternative solution

Now, let's see another way to solve this problem with slightly better time complexity. In this approach, we focus on choosing the left bound for binary search such that the search space reduces to `n - k`.

We initialize the `left` pointer to 0 and the `right` pointer to `nums.length - k`. These values are assigned based on the observation that the left bound can't exceed `nums.length - k` to ensure we have enough elements for the window.

Next, while `left` $<$ `right`, we perform a binary search to find the optimal position for the left bound of the sliding window. We calculate `mid` and compare the distances between `target` and the elements at `nums[mid]` and `nums[mid + k]`. If `|nums[mid] - target|` is greater than `|nums[mid + k] - target|`, it means the element at `nums[mid]` is farther from `target` compared to the element at `nums[mid + k]`. In this case, it updates `left` to `mid + 1`, shifting the left bound to the right. Otherwise, it updates the `right` to `mid`, moving the right bound closer to the left.

Once the while loop completes, return the elements of `nums` starting from `left` and including the next `k` elements. These elements represent the `k` closest elements to `target`.

Since the initial search space has a size of `n-k`, the binary search takes $O(log(n-k))$. Therefore, the time complexity of this solution is $O(log(n-k))$. The space complexity remains $O(1)$.