**Approach**

1. **Sorting**:
   * Both nums1 and nums2 arrays are sorted initially using Arrays.sort(). Sorting helps in efficiently finding common elements using a two-pointer technique.
2. **Two-pointer Technique**:
   * Initialize two pointers, nums1Element and nums2Element, pointing to the beginning of nums1 and nums2 respectively.
   * Use these pointers to traverse through the sorted arrays:
     + If nums1[nums1Element] < nums2[nums2Element], increment nums1Element to move forward in nums1.
     + If nums1[nums1Element] > nums2[nums2Element], increment nums2Element to move forward in nums2.
     + If nums1[nums1Element] == nums2[nums2Element], it means there's an intersection:
       - Copy the intersecting element (nums1[nums1Element]) to nums1[tempElement] (where tempElement keeps track of the number of intersecting elements).
       - Increment both nums1Element and nums2Element.
       - Increment tempElement to prepare for the next intersection element.
3. **Copying Result**:
   * After the loop finishes, use Arrays.copyOfRange(nums1, 0, tempElement) to create a new array containing only the intersecting elements.

**Time Complexity**

* **Sorting**: Arrays.sort() has a time complexity of O(n log n) for each array, where n is the length of the array. Thus, sorting both arrays has a combined time complexity of O(nums1Size log nums1Size + nums2Size log nums2Size).
* **Finding Intersection**:
  + Once sorted, finding the intersection using two pointers (nums1Element and nums2Element) takes O(nums1Size + nums2Size) time because each pointer can traverse through at most all elements of nums1 and nums2 once.
* **Total Time Complexity**:
  + Combining both parts, the dominant term is O(nums1Size log nums1Size + nums2Size log nums2Size) due to sorting.

Therefore, the overall time complexity of the intersect method is **O(nums1Size log nums1Size + nums2Size log nums2Size)**.

**Space Complexity**

* **Sorting**:
  + Sorting typically requires O(1) extra space for arrays, but it can be considered as O(nums1Size + nums2Size) if we consider the space used by the sorting algorithm.
* **Result Array**:
  + The space used for the result array Arrays.copyOfRange(nums1, 0, tempElement) is O(tempElement), which is at most min(nums1Size, nums2Size).

Therefore, the overall space complexity is **O(nums1Size + nums2Size)**, dominated by the space used by sorting and the result array.