<https://leetcode.com/problems/intersection-of-two-arrays-ii/>

Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

**Example 1:**

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2,2]

**Example 2:**

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [4,9]

Explanation: [9,4] is also accepted.

**Constraints:**

* 1 <= nums1.length, nums2.length <= 1000
* 0 <= nums1[i], nums2[i] <= 1000

**Follow up:**

* What if the given array is already sorted? How would you optimize your algorithm?
* What if nums1's size is small compared to nums2's size? Which algorithm is better?
* What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

**Attempt 1: 2023-03-09**

**Solution 1: Hash Table (10 min)**

class Solution {

public int[] intersect(int[] nums1, int[] nums2) {

Map<Integer, Integer> freq1 = new HashMap<Integer, Integer>();

for(int num : nums1) {

freq1.put(num, freq1.getOrDefault(num, 0) + 1);

}

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

for(int num : nums2) {

if(freq1.containsKey(num)) {

result.add(num);

freq1.put(num, freq1.get(num) - 1);

if(freq1.get(num) == 0) {

freq1.remove(num);

}

}

}

return result.stream().mapToInt(i -> i).toArray();

}

}

Time complexity: O(n)

Space complexity: O(n)

**Solution 2: Two Points (10 min)**

class Solution {

public int[] intersect(int[] nums1, int[] nums2) {

Arrays.sort(nums1);

Arrays.sort(nums2);

int i = 0;

int j = 0;

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

while(i < nums1.length && j < nums2.length) {

if(nums1[i] == nums2[j]) {

result.add(nums1[i]);

i++;

j++;

} else if(nums1[i] > nums2[j]) {

j++;

} else {

i++;

}

}

return result.stream().mapToInt(k -> k).toArray();

}

}

Time complexity: O(nlog⁡n)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/intersection-of-two-arrays-ii/solutions/1468295/python-2-approaches-3-follow-up-questions-clean-concise/>

**✔️ Approach 1: HashMap**

* Using HashMap to store occurrences of elements in the nums1 array.
* Iterate x in nums2 array, check if cnt[x] > 0 then append x to our answer and decrease cnt[x] by one.
* To optimize the space, we ensure len(nums1) <= len(nums2) by swapping nums1 with nums2 if len(nums1) > len(nums2).

class Solution:

def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:

if len(nums1) > len(nums2): return self.intersect(nums2, nums1)

cnt = Counter(nums1)

ans = []

for x in nums2:

if cnt[x] > 0:

ans.append(x)

cnt[x] -= 1

return ans

Complexity:

* Time: O(M + N), where M <= 1000 is length of nums1 array, N <= 1000 is length of nums2 array.
* Space: O(min(M, N))

**✔️ Approach 2: Sort then Two Pointers**

class Solution:

def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:

nums1.sort()

nums2.sort()

ans = []

i = j = 0

while i < len(nums1) and j < len(nums2):

if nums1[i] < nums2[j]:

i += 1

elif nums1[i] > nums2[j]:

j += 1

else:

ans.append(nums1[i])

i += 1

j += 1

return ans

Complexity:

* Time: O(MlogM + NlogN), where M <= 1000 is length of nums1 array, N <= 1000 is length of nums2 array.
* Extra Space (without counting output as space): O(sorting)

**✔️ Follow-up Question 1:** What if the given array is already sorted? How would you optimize your algorithm?

* **Approach 2** is the best choice since we skip the cost of sorting.
* So time complexity is O(M+N) and the space complexity is O(1).

**✔️ Follow-up Question 2:** What if nums1's size is small compared to nums2's size? Which algorithm is better?

* **Approach 1** is the best choice.
* Time complexity is O(M+N) and the space complexity is O(M), where M is length of nums1, N is length of nums2.

**✔️ Follow-up Question 3:** What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

If nums1 fits into the memory, we can use **Approach 1** which stores all elements of nums1 in the HashMap. Then, we can sequentially load and process nums2.

If neither nums1 nor nums2 fits into the memory, we split the numeric range into numeric sub-ranges that fit into the memory.

We modify **Approach 1** to count only elements which belong to the given numeric sub-range.

We process each numeric sub-ranges one by one, util we process all numeric sub-ranges.

For example:

Input constraint:

1 <= nums1.length, nums2.length <= 10^10.

0 <= nums1[i], nums2[i] < 10^5

Our memory can store up to 1000 elements.

Then we split numeric range into numeric sub-ranges [0...999], [1000...1999], ..., [99000...99999], then call **Approach 1** to process 100 numeric sub-ranges.

**Refer to**

<https://leetcode.com/problems/intersection-of-two-arrays-ii/solutions/282372/java-solution-with-all-3-follow-up-questions/>

**Follow-up Question 1:** **What if the given array is already sorted? How would you optimize your algorithm?**

Classic two pointer iteration, i points to nums1 and j points to nums2. Because a sorted array is in ascending order, so if nums1[i] > nums[j], we need to increment j, and vice versa. Only when nums1[i] == nums[j], we add it to the result array. Time Complexity O(max(N, M)). Worst case, for example, would be nums1 = {100}, and nums2 = {1, 2, ..., 100 }. We will always iterate the longest array. The example code is below(I sorted it so it could go through OJ):

**Follow-up Question 2: What if nums1's size is small compared to nums2's size? Which algorithm is better?**

This one is a bit tricky. Let's say nums1 is K size. Then we should do **binary search** for every element in nums1. Each lookup is O(log N), and if we do K times, we have O(K log N).If K this is small enough, O(K log N) < O(max(N, M)). Otherwise, we have to use the previous two pointers method. let's say A = [1, 2, 2, 2, 2, 2, 2, 2, 1], B = [2, 2]. For each element in B, we start a binary search in A. To deal with duplicate entry, once you find an entry, all the duplicate element is around that that index, so you can do linear search scan afterward.

Time complexity, O(K(logN) + N). Plus N is worst case scenario which you have to linear scan every element in A. But on average, that shouldn't be the case. so I'd say the Time complexity is O(K(logN) + c), c (constant) is number of linear scan you did.

<https://leetcode.com/problems/intersection-of-two-arrays-ii/solutions/1808056/java-binary-search/>

/\*

nums1 = [1,2,2,1], nums2 = [2,2]

1,1,2,2

2,2

nums1 = [4,9,5], nums2 = [9,4,9,8,4]

4,4,8,9,9

4,5,9

--

main idea:

using binary search, search for the elements of the smallest array (nums1) in the largest array (nums2)

sort the largest array so that binary search is feasible

sort the smallest array so that we can seach sequentially

if element is found,

keep searching to the left until we find the first occurrence of the element

add element to the result

when element is found, keep track of the last index where element was found so that next binary search ignores previous used indexes

ie. nums1 = 1,1 nums2 = 1,2,2 - output should be [1] - once we found first 1 at index 0 and next search is done as of index 1

\*/

public int[] intersect(int[] nums1, int[] nums2) {

if(nums2.length < nums1.length){

return intersect(nums2, nums1);

}

Arrays.sort(nums1);

Arrays.sort(nums2);

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

int leftIndex = 0;

for(int num: nums1){

int index = binarySearch(nums2, num, leftIndex);

if(index != -1){

result.add(num);

leftIndex = index + 1;

}

}

return result.stream().mapToInt(Integer::intValue).toArray();

}

private int binarySearch(int[] nums, int target, int left){

int right = nums.length - 1;

int index = -1;

while(left <= right){

int middle = left + (right - left) / 2;

if(nums[middle] == target){

index = middle;

right = middle - 1;

} else if(nums[middle] > target){

right = middle - 1;

} else {

left = middle + 1;

}

}

return index;

}

**Follow-up Question 3: What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?**

<https://leetcode.com/problems/intersection-of-two-arrays-ii/solutions/1468295/python-2-approaches-3-follow-up-questions-clean-concise/comments/1273113>

This is a highly accepted answer for follow up questions 3:

It's by using HDFS (Hadoop). Basically, it splits your data into multiple CPUs by key - in our case, the key here is the number itself and it's count.

i.e, (number, count). It's just like a counter/hashmap but in multiple CPU's.

hashMap = (file1,file2).map(number => (number,1))

hashMap.reduceByKey(lambda a,b : a+b)

Line 1 converts number into it's own key . So A: [1,2,2,3]AND B: [1,1]become A: (1,1), (2,1), (2,1) (3,1)and B: (1,1), (1,1)

Line 2 combines two similar keys and add's the values.