# Question

Given an integer array arr, count how many elements x there are, such that x + 1 is also in arr.

If there're duplicates in arr, count them seperately.

**Example 1:**

**Input:** arr = [1,2,3]

**Output:** 2

**Explanation:** 1 and 2 are counted cause 2 and 3 are in arr.

**Example 2:**

**Input:** arr = [1,1,3,3,5,5,7,7]

**Output:** 0

**Explanation:** No numbers are counted, cause there's no 2, 4, 6, or 8 in arr.

**Example 3:**

**Input:** arr = [1,3,2,3,5,0]

**Output:** 3

**Explanation:** 0, 1 and 2 are counted cause 1, 2 and 3 are in arr.

**Example 4:**

**Input:** arr = [1,1,2,2]

**Output:** 2

**Explanation:** Two 1s are counted cause 2 is in arr.

**Example 5:**

**Input:** arr = [1,1,2]

**Output:** 2

**Explanation:** Both 1s are counted because 2 is in the array.

**Constraints:**

* 1 <= arr.length <= 1000
* 0 <= arr[i] <= 1000

  Hide Hint #1

Use hashset to store all elements.

Hide Hint #2

Loop again to count all valid elements.

# Solution

## **Solution**

#### **Approach 1: Search with Array**

**Intuition**

The simplest way of solving this problem is to loop through each integer, x, checking whether or not it should be counted. This requires checking whether or not x + 1 is in arr.

define function count\_elements(arr):

count = 0

for each x in arr:

if integer\_in\_array(arr, x + 1):

count = count + 1

return count

To implement the integer\_in\_array function in the above algorithm, we can use **linear search**. To do a linear search, we need to loop through each integer of arr. If we find the integer that we're looking for, then return true. If we get to the end of arr, then we know the integer is not there, and so should return false.

define function integer\_in\_array(arr, target):

for each x in arr:

if target is equal to x:

return true

return false

Many programming languages have a built in function for checking whether or not an integer is in arr, e.g. Python.

**Algorithm**

|  |
| --- |
| class Solution {  public int countElements(int[] arr) {  int count = 0;  for (int x : arr) {  if (integerInArray(arr, x + 1)) {  count++;  }  }  return count;  }  public boolean integerInArray(int[] arr, int target) {  for (int x : arr) {  if (x == target) {  return true;  }  }  return false;  }  } |

|  |
| --- |
| def countElements(self, arr: List[int]) -> int:  count = 0  for x in arr:  if x + 1 in arr:  count += 1  return count  # Note that we could also do this as a one-liner generator comprehension.  # return sum(1 for x in arr if x + 1 in arr) |

**Complexity Analysis**

Let N*N* be the length of the input array, arr.

* Time complexity : O(N^2)*O*(*N*2).

We loop through each of the N*N* integers x, checking whether or not x + 1 is also in arr. Checking whether or not x + 1 is in arr is done using linear search, which requires checking through all N*N* integers in arr. Because we're doing N*N* operations N*N* times, we get a time complexity of O(N^2)*O*(*N*2).

* Space complexity : O(1)*O*(1).

We are only using a constant number of single-value variables (e.g. count), giving us a space complexity of O(1)*O*(1).

#### **Approach 2: Search with HashSet**

**Intuition**

If you're not familiar with the HashSet data structure, check out our [Hash Tables Explore Card](https://leetcode.com/explore/learn/card/hash-table/) to get up to speed.

The above algorithm will work fine for the maximum array length we're given here. However, we can do a lot better than O(N^2)*O*(*N*2), and an interviewer will no doubt expect you to come up with a better way.

The reason why the algorithm above was so inefficient is because we're performing N*N* linear searches, each with a cost of O(N)*O*(*N*). When we have an algorithm that is performing many linear searches to check for item existence, we should instead be looking to change the way the data is stored so that the time complexity of doing each search is less.

Recall that looking up items in a HashSet has a cost of O(1)*O*(1). Creating a HashSet from an array of N*N* items has a cost of O(N)*O*(*N*). We only need to create the HashSet once. After that, we can then replace all O(N)*O*(*N*) linear searches with O(1)*O*(1) HashSet lookups.

Before we go any further, here is an algorithm that is incorrect. Try to spot what the problem is; we'll discuss it just below.

define function count\_elements(arr):

hash\_set = a new HashSet

add all integers of arr to hash\_set

count = 0

for each x in hash\_set:

if hash\_set contains x + 1:

count = count + 1

return count

Did you spot the bug? If there were duplicates in arr, then the count returned will be too low!

Recall that a HashSet removes duplicates. Consider a case like arr = [1, 1, 2]. The HashSet will be {1, 2}. Therefore, the above code will loop over each integer in the HashSet, which is only one copy of 1. Yet arr had two copies of 1.

To fix it, we need to loop over arr, but do the existence checks using the HashSet.

define function count\_elements(arr):

hash\_set = a new HashSet

add all integers of arr to hash\_set

count = 0

for each x in arr:

if hash\_set contains x + 1:

count = count + 1

return count

**Algorithm**

|  |
| --- |
| public int countElements(int[] arr) {  Set<Integer> hashSet = new HashSet<>();  for (int x : arr) {  hashSet.add(x);  }  int count = 0;  for (int x : arr) {  if (hashSet.contains(x + 1)) {  count++;  }  }  return count;  } |

|  |
| --- |
| def countElements(self, arr: List[int]) -> int:  hash\_set = set(arr)  count = 0  for x in arr:  if x + 1 in hash\_set:  count += 1  return count |

**Complexity Analysis**

Let N*N* be the length of the input array, arr.

* Time complexity : O(N)*O*(*N*).

Creating a HashSet from N*N* integers takes O(N)*O*(*N*) time. We then need to loop over each of the N*N* integers like before, except this time we check for x + 1 by seeing if it is in the HashSet; an O(1)*O*(1) operation. This gives us a total time complexity of O(N) + N \cdot O(1) = O(N) + O(N) = O(N)*O*(*N*)+*N*⋅*O*(1)=*O*(*N*)+*O*(*N*)=*O*(*N*).

* Space complexity : O(N)*O*(*N*).

The HashSet needs to store each unique integer from arr. In the worst case, all the integers in arr will be unique, meaning that the HashSet has a space complexity of O(N)*O*(*N*).

It's interesting to note that O(N)*O*(*N*) is an upper bound on the space complexity. If U*U* is the number of unique integers in arr, then the space complexity could more accurately be represented as O(U)*O*(*U*).

#### **Approach 3: Search with Sorted Array**

**Intuition**

Another way of changing the data storage to allow for more efficient searching is to sort it. Sorting has a time complexity of O(N \, \log \, N)*O*(*N*log*N*), and searching for integers in a sorted array, using binary search, has a cost of O(\log \, N)*O*(log*N*). This will give us a total time complexity of O(N \, \log \, N)*O*(*N*log*N*).

define function countElements(arr):

sort arr

count = 0

for each x in arr:

binary search for x + 1 in arr

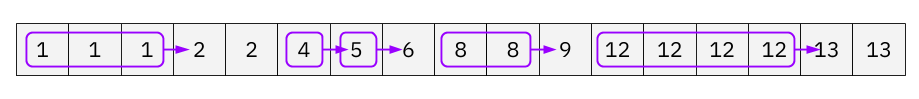
if x + 1 is in arr:

count = count + 1

return count

The main challenge of this approach would be needing to implement your own binary search.

However, we don't actually need to use binary search! If we iterate over the sorted arr, then we know that if x + 1 exists, it will be after all the copies of x.



Each copy of x should be counted if at least one copy of x + 1 exists. Therefore, we can iterate down the sorted arr, keeping track of how many times the current x has appeared. When we get to a different integer, we can check if it's x + 1, and if it is, then the number of x we saw should be added to count.

define function countElements(arr):

sort arr

count = 0

run\_length = 1

for each i in range 1 to arr.length - 1:

if arr[i - 1] is not equal to arr[i]:

if arr[i - 1] + 1 is equal to arr[i]:

count = count + run\_length

run\_length = 0

run\_length = run\_length + 1

return count

Here is an animation of this approach.

|  |
| --- |
| public int countElements(int[] arr) {  Arrays.sort(arr);  int count = 0;  int runLength = 1;  for (int i = 1; i < arr.length; i++) {  if (arr[i - 1] != arr[i]) {  if (arr[i - 1] + 1 == arr[i]) {  count += runLength;  }  runLength = 0;  }  runLength++;  }  return count;  } |

|  |
| --- |
| def countElements(self, arr: List[int]) -> int:  arr.sort()  count = 0  run\_length = 1  for i in range(len(arr)):  if arr[i - 1] != arr[i]:  if arr[i - 1] + 1 == arr[i]:  count += run\_length  run\_length = 0  run\_length += 1  return count |

**Complexity Analysis**

* Time complexity : O(N \, \log \, N)*O*(*N*log*N*).

Sorting using a built-in sorting algorithm has a cost of O(N \, \log \, N)*O*(*N*log*N*). After that, we do a single pass through arr, which has a cost of O(N)*O*(*N*), giving us a total time complexity of O(N \, \log \, N) + O(N) = O(N \, \log \, N)*O*(*N*log*N*)+*O*(*N*)=*O*(*N*log*N*).

* Space complexity : varies from O(N)*O*(*N*) to O(1)*O*(1).

The space complexity of this approach is dependent on the space complexity of the sorting algorithm you're using. The space complexity of sorting algorithms built into programming languages is generally anywhere from O(N)*O*(*N*) to O(1)*O*(1).

Notice that you could implement your own O(N \, \log \, N)*O*(*N*log*N*) time complexity, O(1)*O*(1) space complexity, sorting algorithm if needed. In practice, O(N \, \log \, N)*O*(*N*log*N*) is not much worse than O(N)*O*(*N*), and so this approach provides an interesting contrast to Approach 2 (which had a space complexity of O(N)*O*(*N*)).