Arden Dertat
Information Retrieval and Machine Learning

Programming Interview Questions 4: Find Missing Element

Posted on September 27, 2011 by Arden

This question can be solved efficiently with a very clever trick. There is an array of non-negative integers. A second array is formed by shuffling the elements of the first array and deleting a random element. Given these two arrays, find which element is missing in the second array. Here is an example input, the first array is shuffled and the number 5 is removed to construct the second array.

First Array = [4, 1, 0, 2, 9, 6, 8, 7, 5, 3]

Second Array = [6, 4, 7, 2, 1, 0, 8, 3, 9]

The naive way to solve it is for every element in the second array, check whether it appears in the first array. Note that there may be duplicate elements in the arrays so we should pay special attention to it. The complexity of this approach is O(N^2). A more efficient solution is to sort the first array, so while checking whether an element in the first array appears in the second, we can do binary search. But we should still be careful about duplicate elements. The complexity is O(NlogN). If we don't want to deal with the special case of duplicate numbers, we can sort both arrays and iterate over them simultaneously. Once two iterators have different values we can stop. The value of the first iterator is the missing element. This solution is also O(NlogN). Here is the algorithm for this approach:

```
def findMissingNumber1(array1, array2):
    array1.sort()
    array2.sort()
    for num1, num2 in zip(array1, array2):
        if num1!=num2:
            return num1
    return array1[-1]
```

Let's see whether we can do better. In most interviews, we would be expected to come up with a linear time solution. We can use a hashtable and store the number of times each element appears in the second array. Then for each element in the first array we decrement its counter. Once hit an element with zero count that's the missing element. Here is the algorithm:

```
def findMissingNumber2(array1, array2):
    d=collections.defaultdict(int)
    for num in array2:
        d[num]+=1
    for num in array1:
        if d[num]==0:
            return num
        else:
            d[num]-=1
```

The time complexity is optimal O(N) but the space complexity is also O(N), because of the hashtable. Ideally we would like to have constant space complexity. One possible solution is computing the sum of all the numbers in array1 and array2, and subtracting array2's sum from array1's sum. The difference is the missing number in array2. However, this approach is somewhat problematic. What if the arrays are too long, or the

numbers are very large. Then overflow will occur while summing up the numbers.

By performing a very clever trick, we can achieve linear time and constant space complexity without any problems. Here it is: initialize a variable to 0, then XOR every element in the first and second arrays with that variable. In the end, the value of the variable is the result, missing element in array2. Classy, isn't it? Here is the code:

```
def findMissingNumber3(array1, array2):
    result=0
    for num in array1+array2:
        result^=num
    return result
```

Let's analyze why this approach works. What happens when we XOR two numbers? We should think bitwise, instead of decimal. XORing a 4-bit number with 1011 would flip the first, third, and fourth bits of the number. XORing the result again with 1011 would flip those bits back to their original value. So, if we XOR a number two times with some number nothing will change. We can also XOR with multiple numbers and the order would not matter. For example, say we XOR the number n1 with n2, then XOR the result with n3, then XOR their result with n2, and then with n3. The final result would be the original number n1. Because every XOR operation flips some bits and when we XOR with the same number again, we flip those bits back. So the order of XOR operations is not important. If we XOR a number with some number an odd number of times, there will be no effect.

Above we XOR all the numbers in array1 and array2. All numbers in array2 also appear in array1, but there is an extra number in array1. So the effect of each XOR from array2 is being reset by the corresponding same number in array1 (remember that the order of XOR is not important). But we can't reset the XOR of the extra number in array1, because it doesn't appear in array2. So the result is as if we XOR 0 with that extra number, which is the number itself. Since XOR of a number with 0 is the number. Therefore, in the end we get the missing number in array2. The space complexity of this solution is constant O(1) since we only use one extra variable. Time complexity is O(N) because we perform a single pass from the arrays.

This question demonstrates the power of bitwise operations, and how to use them effectively to achieve optimal solutions. It's one of my favorite interview questions.

27 comments

★ 2

Leave a message...

Best Community Share

Avatar vs • 2 years ago
Hey Arden,

Your solutions to the problem are brilliant! I like how you analyze various approaches and slowly lead the reader to the optimal solution (rather than presenting the reader with the best solution at the beginning).

Keep up the good work and thanks so much for sharing!
8 Reply Share ›

anjali → vs • a year ago
I agree! Thank you so much!
1 Reply Share ›

Avatar Petar • 11 months ago
There is another simple O(N) solution. We only need to sum all the elements of the first array, and than subtract the sum of elements of the second array. The result will be the missing number. Only problem here (same as the last solution presented) is that it gives the same result in the case of the missing number 0, and the case when there is no missing number.
3 Reply Share ›

rahul → Petar • 10 months ago
Sum may hit the overflow condition. so i feel it is better to go with XOR approatch
2 Reply Share ›

Jinesh → Petar • 4 months ago
An if condition checking the length of both the arrays at the start will avoid this issue..
Reply Share ›

Avatar Ashot Madatyan • a year ago
I woul like to add a special note for the "XOR" solution: this will work iff the only duplicate value is not zero.
3 Reply Share ›

Avatar Sriram • 4 months ago
Arden, How about add all the elements of array1 and call it sum1, next add all the elements of array2 and call it sum2, sum1-sum2 gives the missing element :)
2 Reply Share ›

Avatar ANONYMOUS • 2 years ago
another interesting question: two numbers are missing.
1 Reply Share ›

Arden → ANONYMOUS • 2 years ago
Yes that's also a good one involving some math. I can write about that as well, thanks for the advice.
1 Reply Share ›

libo → Arden • a year ago
is it possible to post the code for missing two elements? thanks for sharing in advance, and i forget the advanced math after graduation.
Reply Share ›

Avatar MdT • 6 months ago
Quick question, why do you say "If we XOR a number with some number an odd number of times, there will be no effect."

As you said, n1 XOR'd with n2 twice would give n1. If XOR'd 4 times, it would give n1 again. Then why "odd number of times"?

Reply Share ›

Avatar

Funsi • 6 months ago

How about
missing_entry = sum(array1) - sum(array2)

??

Reply Share ›

Avatar

Petar • 11 months ago

*o

Reply Share ›

Avatar

Lucas • a year ago

Your xor solution actually uses linear space since Python will actually construct the the array array1+array2 in the for loop. You should be able to get around this either by using itertools.chain or by splitting into two for loops.

Reply Share ›

Avatar

Harsh • a year ago

Brilliant !!

Reply Share ›

Avatar

Steve • a year ago

If you used unsigned integers for the sum then there is no problem with overflow is there? The value just "wraps" and the end result is correct.

Reply Share ›

Avatar

Anonymous • a year ago

```
// c++/java code
int findMissing(int a[], int b[], int aSize, int bSize) {

int result = 0;
int i=0; int j=0;
for (i=0, j=0; i<aSize; i++) {
result ^= a[i];
if (j<bSize) {
result ^= b[j++];
}
}
return result;
}
```

Reply Share ›

Avatar

Ashot Madatyan • 2 years ago

Ապրես, շատ սիրուն լուծումներ ես առաջարկում, Արդեն:

Reply Share ›

Avatar

s • 2 years ago

Hi Arden,

What about an alternative for the "adding up" solution as follows. Instead of summing the first and second array, then subtracting from each other, which can cause overflow, we can interleave between adding and subtracting. That is,

-take the 1st element in the 1st array
- subtract it from the 1st element in the 2nd,
- add to the 2nd element in the 1st array,
- subtract the result from the 2nd element in the 2nd array.
- so on.

We have an O(n) in time and O(1) in space. Let me know what you think. Thanks for your wonderful blog Ardan.

Reply Share ›

Arden ➔ s • 2 years ago

Thanks a lot for the comment. This is definitely a better solution than adding up, but it may also cause overflow. Imagine large numbers being in front of array1 and small numbers being in the end. And after shuffling array2 contains small numbers in front and large numbers move to the end. Then while subtracting and adding the sum will continually increase, which may lead to overflow. But this is the extreme worst case of course, and probably very unlikely to happen if we have a good shuffle function. Therefore, your approach is much better than simply adding up two arrays first and then subtracting. Because now the intermediate sum increases much slowly. Thanks for noticing. I hope you enjoy the blog..

Reply Share ›

t ➔ Arden • 2 years ago

just balance the residue around 0, i.e. keeps subtracting until it's below 0, then adding back until it goes above 0.

Reply Share ›

melih ➔ t • 11 months ago

This will add more complexity (additional if checks), I think the usage of XOR is the most brilliant way to solve this problem. Great blog Arden and it is very helpful, Teşekkürler:)

1 Reply Share ›

Avatar ahmet alp balkan • 2 years ago

Peki ya xor yerine sum kullanirsak herhangi bir drawback'i olur mu sence?

Reply Share ›

Arden ➔ ahmet alp balkan • 2 years ago

Python'da sorun olmayacaktir. Cunku "Python seamlessly converts a number that becomes too large for an integer to a long. And long integers have unlimited precision". Ama Python'da uzun listelerde XOR kisa listelerde sum daha hizli calisiyo.

Reply Share ›

Avatar ahmet alp balkan • 2 years ago

Gercekten guzeldi :)

Reply Share ›

ege ➔ ahmet alp balkan • 2 years ago

Arden güzel soru - güzel cevap ancak tüm elemanları toplama fikri daha sade bence
kod yazarken senden sonra bakacak adamın anlayabilmesi de çok önemli bir proje için :)

Reply Share ›

Arden ➔ ege • 2 years ago

Haklisin Ege okunabilirlik cok onemli tabi, ama interview'da asil amac en optimize sekilde cozmek. Hatta interviewer'in gormedigi orjinal guzel bir cozum uretirsen senden iyisi yok..

Reply Share ›

Subscribe Add Disqus to your site

Arden Dertat
Proudly powered by WordPress.