That XOR Trick

← Home

March 15, 2020

There are a whole bunch of popular interview questions that can be solved in one of two ways: Either using common data structures and algorithms in a sensible manner, or by using some properties of XOR in a seemingly hard to understand way.

While it seems unreasonable to expect the XOR solutions in interviews, it is quite fun to figure out how they work. As it turns out, they are all based on the same fundamental trick, which we will derive in a bottom-up way in this post. Afterwards we will look at a bunch of applications of *that XOR trick* ™, such as solving this popular interview question:

> *You are given an array of n - 1 integers which are in the range between 1 and n. All numbers appear exactly once, except one number, which is missing. Find this missing number.*

Of course, there are a number of straightforward ways to solve this problem, but there is also a perhaps surprising one using XOR.

XOR

XOR is a logical operator that works on bits. Let's denote it by `^`. If the two bits it takes as input are the same, the result is `0`, otherwise it is `1`. This implements an *exclusive or* operation, i.e. exactly one argument has to be `1` for the final result to be `1`. We can show this using a truth table:

x	y	x ^ y
0	0	0

x	y	x ^ y
0	1	1
1	0	1
1	1	0

Most programming languages implement `^` as a bitwise operator, meaning XOR is individually applied to each bit in a string of bits (e.g. a byte).

For example:

```
0011 ^ 0101 = 0110
```

since

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

Because of this, we can apply XOR to anything, not just booleans.

Deducing Some Useful Properties

We can derive a bunch of properties from the previous definition. Let's go through them one by one, and then compose them to solve the interview questions previously mentioned.

XOR and 0: `x ^ 0 = x`

If one of the two arguments to XOR is `0`, then the remaining argument is the result. This directly follows from the truth table by checking the rows where `y = 0`, namely the first and third row.

x	y	x ^ y
0	**0**	**0**
0	1	1
1	**0**	**1**
1	1	0

XOR on the same argument: `x ^ x = 0`

If the two arguments are the same, the result is always `0`. Again, we can convince ourselves that this is true by inspecting the truth table. This time we have to check the rows where `x = y`, i.e. the first and last row.

x	y	x ^ y
0	**0**	**0**
0	1	1
1	0	1
1	**1**	**0**

Intuitively, this means that if we apply XOR to the same arguments, they cancel each other out.

Commutativity: `x ^ y = y ^ x`

XOR is commutative, meaning we can change the order in which we apply XOR. To prove this, we can check the truth table for both `x ^ y` and `y ^ x`:

x	y	x ^ y	y ^ x
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

As we can see, `x ^ y` and `y ^ x` always end up with the same value.

Sequences of XOR operations

By combining all of this, we can deduce the central insight behind everything that is about to follow:

> ***The XOR trick**: If we have a sequence of XOR operations `a ^ b ^ c ^ ...`, then we can remove all pairs of duplicated values without affecting the result.*

Commutativity allows us to re-order the applications of XOR so that the duplicated elements are next to each other. Since `x ^ x = 0` and `a ^ 0 = a`, each pair of duplicated values has no effect on the outcome.

Let's go through an example of this:

```
  a ^ b ^ c ^ a ^ b     # Commutativity
= a ^ a ^ b ^ b ^ c     # Using x ^ x = 0
= 0 ^ 0 ^ c             # Using x ^ 0 = x (and commutativity)
= c
```

Because `^` is a bitwise operator, this will work regardless of what kind of values `a`, `b` and `c` are. This idea is really at the heart of how XOR can be used seemingly magically in many situations.

Application 1: In-Place Swapping

Before we solve the problem of finding the missing number, let's start with this simpler problem:

> *Swap two values x and y in-place, i.e. without using any helper variables.*

It turns out that one can easily solve this problem using the following three XOR instructions:

```
x ^= y
y ^= x
x ^= y
```

That seems a bit mysterious. Why do we end up swapping `x`, `y` if we do this?

To see how it works, let's go through this step by step. The comment after each instruction shows the current values of `(x, y)`:

```
x ^= y # =>                        (x ^ y, y)
y ^= x # => (x ^ y, y ^ x ^ y) = (x ^ y, x)
x ^= y # => (x ^ y ^ x, x)     = (y, x)
```

We can see that this is really just using the properties derived earlier.

The fundamental insight here is that having `x ^ y` in one register and `x` in the other, allows us to perfectly reconstruct `y`. Once `x ^ y` is stored (instruction 1),

we can just put `x` into the other register (instruction 2), and then use it to change `x ^ y` to just `y` (instruction 3).

Application 2: Finding the Missing Number

Let's finally solve the problem posed in the beginning of this post:

> *You are given an array `A` of n - 1 integers which are in the range between 1 and n. All numbers appear exactly once, except one number, which is missing. Find this missing number.*

Of course, there are many straightforward ways of solving this, but we did set out to do it using XOR.

From the XOR trick we know that having a sequence of XOR statements means we can remove all duplicated arguments. If we just XOR all values in the given list, however, we cannot apply this trick because there are no duplicates:

```
A[0] ^ A[1] ^ ... ^ A[n - 2]
```

Note that `A[n - 2]` is the last index of a list of `n - 1` elements.

What we can do additionally is to also XOR all values between 1 and n:

```
1 ^ 2 ^ ... ^ n ^ A[0] ^ A[1] ^ ... ^ A[n - 2]
```

This will give us a sequence of XOR statements where elements appear as follows:

- All values in the given list now appear twice:
  - once from taking all the values between 1 and n
  - once because they were in the original list

- The missing value appears exactly once:
  - once from taking all the values between 1 and n

If we XOR all of this, we essentially remove all values that appear twice, thanks to the XOR trick. This means that we are left with the missing value, which happens to be exactly what we were looking for in the first place.

Coded up, it looks something like this:

```
def find_missing(A, n):
  result = 0

  # XOR of all the values from 1 to n
  for value in range(1, n + 1):
    result ^= value

  # XOR of all values in the given array
  for value in A:
    result ^= value

  return result
```

Just looking at the code, this seems to be a difficult to understand algorithm. When knowing how the XOR trick works, however, it becomes fairly trivial. I think that also shows why it is unreasonable to expect this solution in an interview: It requires knowledge of a very specific trick but not much algorithmic thinking beyond that.

Before we move on to the next application, let me follow up with two remarks.

Generalizing this beyond integers

While we worked on integers from 1 to n so far, this is not required. In fact, the previous algorithm works in any situation where there is (1) some set of potential

elements and (2) a set of elements actually appearing. The sets may only differ in the one missing element. This worked out nicely for integers because the set of potential elements just corresponds to the elements from 1 to n.

One could imagine applications where the elements are not integers from 1 to n:

- The set of potential elements are `Person` objects and we ought to find the `Person` missing from a list of values
- The set of potential elements are all nodes in a graph and we are looking for a missing node
- The set of potential elements are integers in general (not necessarily from 1 to n) and we want to find a missing integer

Arithmetic operators instead of XOR

If the algorithm still seems a bit magical – which I hope it does not – it might help to think about how one could achieve the same result using arithmetic operators. This is actually fairly straightforward:

```
def find_missing(A, n):
  result = 0

  # Add all the values from 1 to n
  for value in range(1, n + 1):
    result += value

  # Subtract all values in the given array
  for value in A:
    result -= value

  return result
```

We add all potential integers and then subtract the ones actually appearing. The solution is not as nice because one would need to handle overflows and because it requires the type of the elements to support `+`, `-` with certain properties. It, however, has the same logic of elements canceling each other out because they appear a certain number of times (once positive, once negative).

Application 3: Finding the Duplicate Number

Here's where it gets fun: We can apply the *exact* same solution to a similar interview question:

> *You are given an array* `A` *of* $n + 1$ *integers which are in the range between 1 and n. All numbers appear exactly once, except one number, which is* ***duplicated****. Find this duplicated number.*

Let's think about what happens if we just apply the exact same algorithm as in the previous solution. We would get a sequence of XOR statements where elements appear as follows:

- The duplicated value appears three times:
  - once from taking all the values between 1 and n
  - twice because it was duplicated in the original list
- All other values in the given list appear twice:
  - once from taking all the values between 1 and n
  - once because they were unique in the original list

As previously, all the duplicated elements cancel each other out. This means we are left with exactly what we are looking for: the element that was duplicated in the original array. This element appearing three times, combined with XOR, reduces to that very element:

```
  x ^ x ^ x
= x ^ 0
= x
```

All other elements cancel each other out because they appear exactly twice.

Application 4: Finding Two Missing/Duplicate Numbers

It turns out we can take this even further. Consider the following, slightly more difficult, problem:

> *You are given an array* `A` *of* ***n - 2*** *integers which are in the range between 1 and n. All numbers appear exactly once, except* ***two*** *numbers, which are missing. Find these two missing numbers.*

As before, the problem is completely equivalent when looking for two *duplicated* numbers, instead of two *missing* numbers.

And I am sure you guessed it, but we will stick with what worked before and start exactly the same way: Let's consider what happens if we use the previous XOR algorithm. If we do that, we again end up with a sequence of XOR statements where all elements cancel each other out, except the two we are looking for.

We will denote these elements by `u` and `v`, mostly because we have not used those letters before. After applying the previous algorithm, we are thus left with `u ^ v`. What can we do with that? We somehow need to extract `u` and `v` from this value, but it is not immediately clear to do that.

Partitioning based on inspecting `u ^ v`

Luckily, we can figure out what to do by using what we already stated earlier. Let's think about this:

> *If the two bits XOR takes as input are the same, the result is `0`, otherwise it is `1`.*

If we analyze the individual bits in `u ^ v`, then every `0` means that the bit had the same value in both `u` and `v`. Every `1` means that the bits differed.

Using this, we find the first `1` in `u ^ v`, i.e. the first position `i` where `u` and `v` have to differ. Then we partition `A` as well as the numbers from 1 to n according to that bit. We end up with two partitions, each of which contains two sets:

1. Partition `0`
    1. The set of all values from 1 to n where the `i`-th bit is `0`
    2. The set of all values from A where the `i`-th bit is `0`
2. Partition `1`
    1. The set of all values from 1 to n where the `i`-th bit is `1`
    2. The set of all values from A where the `i`-th bit is `1`

Since `u` and `v` differ in position `i`, we know that they have to be in different partitions.

Reducing the problem

Next, we can use another insight described earlier:

> *While we worked on integers from 1 to n so far, this is not required. In fact, the previous algorithm works in any situation where there is (1) some set of potential elements and (2) a set of elements actually appearing. The sets may only differ in the one missing (or duplicated) element.*

These two sets correspond exactly to the sets we have in each partition. We can thus search for `u` by applying this idea to one of the partitions and finding the missing element, and then find `v` by applying it to the other partition.

This is actually a pretty nice way of solving it: We effectively reduce this new problem to the more general version of the problem we solved earlier.

Reaching the Limit

One might try to take this further and aim to solve the problem for more than two missing values. I did not give this an abundance of thought, but I think this is where we stop succeeding with XOR. If more than two elements are missing (or duplicated), then analyzing the individual bits fails because there are several combinations possible for both `0` and `1` as results.

The problem then seems to require more complex solutions, which are not based on XOR anymore.

Final Thoughts

As mentioned before, interview questions based on this trick do not seem like a great idea. They require knowing a slightly obscure trick, but once that trick is known, there is not much left to solve (except maybe for application 4). There is also barely a way to show algorithmic thinking (other than reduction) and no good way to make use of data structures.

However, I found it pretty cool to find out how this trick actually work. XOR seems to have just the right properties for all of this to work out. It is also kind of beautiful that something as fundamental as XOR can be used to build up all the things described here.

Thanks to Eugen for the discussions that lead to this post. It was fun to figure out together how all of this works.

Other Posts

What I read in 2020 December 30, 2020

Diffing December 29, 2020

What I read in 2019 December 30, 2019