Your company delivers breakfast via autonomous quadcopter drones. And something mysterious has happened.

Each breakfast delivery is assigned a unique ID, a positive integer. When one of the company's 100 drones takes off with a delivery, the delivery's ID is added to an array, `delivery_id_confirmations`. When the drone comes back and lands, the ID is again added to the same array.

After breakfast this morning there were only 99 drones on the tarmac. One of the drones never made it back from a delivery. **We suspect a secret agent from Amazon placed an order and *stole* one of our patented drones**. To track them down, we need to find their delivery ID.

Given the array of IDs, which contains many duplicate integers and *one unique integer*, find the unique integer.

*The IDs are **not** guaranteed to be sorted or sequential. Orders aren't always fulfilled in the order they were received, and some deliveries get cancelled before takeoff.*

Gotchas

We can do this in $O(n)$ time.

No matter how many integers are in our input array, we can always find the unique ID in $\boldsymbol{O(1)}$ **space!**

Breakdown

A brute force approach would use two nested loops to go through every ID and check every *other* ID to see if there's a duplicate.

This would take $O(n^2)$ time and $O(1)$ space. Can we bring that runtime down?

Well, we know every integer appears twice, except for one integer, which appears once. **Can we just *track* how many times each integer appears?**

We could iterate through the array and **store each integer in a hash**, where the keys are the integers and the values are the number of times we've seen that integer so far. At the end, we'd just need to return the integer we saw one time.

```ruby
def find_unique_delivery_id(delivery_ids)

    ids_to_occurrences = {}

    delivery_ids.each do |delivery_id|
        if ids_to_occurrences.include? delivery_id
            ids_to_occurrences[delivery_id] += 1
        else
            ids_to_occurrences[delivery_id]  = 1
        end
    end

    ids_to_occurrences.each do |delivery_id, occurrences|
        return delivery_id if occurrences == 1
    end
end
```

Alright, we got our runtime down to $O(n)$. That's probably the best runtime we can get—to find our unique integer we'll definitely have to look at *every* integer in the worst case.

But now we've added $O(n)$ space, for our hash. Can we bring that down?

Well, we could use booleans

> **Boolean** is a data type that has only two possible values: true and false.
>
> Because an individual bit also has just two possible values (0 and 1), it's tempting to think that a boolean is stored in just 1 bit. But in practice, booleans often take up a whole byte (8 bits). This is because a byte is usually the **basic addressable unit** in a computer. In other words, usually the computer doesn't grab *individual* bits in memory—instead it grabs "chunks" of 8 bits or more at a time.
>
> Sometimes booleans take up as much space as an integer, which is often 32 or 64 bits in modern computers. In fact, in some languages, like Python, booleans are *subclasses* of ints.

as our values, instead of integers. If we see an integer, we'll add it as a key in our hash with a boolean value of `true`. If we see it again, we'll change its value to `false`. At the end, our non-repeated order ID will be the one integer with a value of `true`.

How much space does this save us? Depends how our language stores booleans vs integers. Often booleans take up just as much space as integers.

And even if each boolean *were* just 1 bit, that'd still be $O(n)$ space overall.

So using booleans probably doesn't save us much space here. Any other ideas?

Let's zoom out and think about what we're working with. The only data we have is integers. How are integers stored?

Our machine stores integers as binary numbers

When we use numbers, we usually use **decimal numbers** (or **base-10**), which are expressed using 10 values, $0 - 9$.

So our digit columns increase by 10 times (1s, 10s, 100s). For example, let's take the digits 101:

base 10

	100s	10s	1s
Digits	1	0	1
Digit column values in base 10	100 +	0 +	1
Value in base 10	= 101		

But using 10 values is arbitrary. We could mulitply our columns by *any* number and numbers would still work. Some languages spoken in Nigeria and India use **duodecimal** numbers, or **base-12**. So "eleven" and "twelve" aren't built using 1s and 2s, they're entirely different digits.

Some mathematicians argue that base-12 is a better system than our base-10, because 12 has more factors (1, 2, 3, 4, 6) than 10 does (1, 2, 5). We probably use decimal numbers because we have 10 fingers.

Binary numbers (or **base-2**) only use two values, 0 and 1. So binary digit columns increase by 2 times (1s, 2s, 4s).

Let's look at the same digits 101:

base 2

	4s	2s	1s
Digits	1	0	1
Digit column values in base 10	4 +	0 +	1
Value in base 10	= 5		

Binary numbers are nice for computers because they can easily be expressed as series of bits, which only have two states (think of them as "on" or "off", "open" or "closed", or 0 or 1).

Here are the base-10 numbers 0 through 10 in binary:

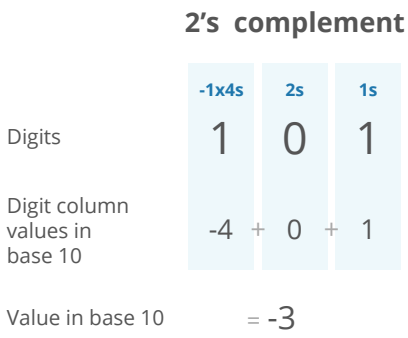
```
decimal  binary

   0      0000
   1      0001
   2      0010
   3      0011
   4      0100
   5      0101
   6      0110
   7      0111
   8      1000
   9      1001
  10      1010
```

Negative numbers are typically represented in binary using *two's complement* encoding. In two's complement, the leftmost digit is *negative*, and the rest of the digits are positive.

Warning: The leftmost digit is *not* the same as a negative sign. The absolute value of the leftmost digit is the same as described above, and each digit's value is double the value of the digit to the right.

To make this clearer, let's look at what happens when we interpret that "101" as two's complement:



> Fun computer systems trivia fact: two's complement isn't the only way negative numbers could be encoded. Other encodings tossed around in the 1960s included "one's complement" and "sign and magnitude" encodings. Of the three encodings, two's complement is the one still used today for a few reasons:
>
> 1. There is only one way to represent zero.
> 2. Basic operations like addition, subtraction, and multiplication are the same regardless of whether the numbers involved are positive or negative.
>
> Since two's complement had both of these properties (and the others didn't), it stuck around and is still used today.

Here are the base-10 numbers -5 through 5 in two's complement, along with how we'd interpret each bit:

```
decimal  binary     interpretation

  -5      1011      -8 + 2 + 1
  -4      1100      -8 + 4
  -3      1101      -8 + 4 + 1
  -2      1110      -8 + 4 + 2
  -1      1111      -8 + 4 + 2 + 1
   0      0000      0
   1      0001      1
   2      0010      2
   3      0011      2 + 1
   4      0100      4
   5      0101      4 + 1
```

> So, should 1011 be read as "eleven" (in binary) or "negative five" (in two's complement)?
>
> It could be either one! Many languages have two types of numbers: **signed** and **unsigned**. Signed numbers are represented in two's complement, and unsigned numbers use plain old base 2.
> So, if an interviewer asks you to convert base-2 into decimal, ask: "is that in two's complement or not?"

using bits.↴

At a deep level, your computer works with bits—`1`s and `0`s. But these get bundled up into more human readable things, like characters and arrays. This is called **abstraction**.

While abstractions are nice, sometimes we want to work directly with bits. You can do this with **bit manipulation**, which involves **bitwise operations**.

Bitwise operations include AND, OR, XOR, NOT, and Bit Shifts.

AND, for example, takes two sets of bits. For each pair of bits (two bits at the same index in each set) it returns `1` *only if both* bits are `1`. Otherwise, it returns `0`. OR would return `1` if *either* of the bits were `1`.

Ruby

```ruby
# AND
1 & 1 # gives 1
0 & 1 # gives 0
0010 & 0111 # gives 0010

# OR
0 | 1 # gives 1
0 | 0 # gives 0
1001 | 0100 # gives 1101
```

What if we thought about this problem on the level of individual bits?

Let's think about the **bitwise operations** AND,↴

The **AND** bitwise operation takes two sets of bits and for each pair of bits (the two bits at the same index in each set) returns `1` if **both** bits are `1`. Otherwise, it returns `0`.

```
1 & 1  →  1
1 & 0  →  0
0 & 1  →  0
0 & 0  →  0
```

Think of it like a hose with two knobs. *Both* knobs must be set to on for water to come out.

When performing AND on two integers, only the digit columns shared by both integers remain:

Ruby

```ruby
5 & 6 # gives 4

# at the bit level:
#     0101  (5)
#   & 0110  (6)
#   = 0100  (4)
```

OR,

The **OR** bitwise operation takes two sets of bits and for each pair of bits (the two bits at the same index in each set) returns 1 if **either** of the bits are 1. Otherwise, it returns 0.

```
1 | 1  →  1
1 | 0  →  1
0 | 1  →  1
0 | 0  →  0
```

Think of it like a bucket with two holes in it. If *both* holes are closed, no water comes out. If *either* hole is open, *or if both* are open, water comes out.

When performing OR on two integers, all digit columns used by either of the integers remain:

Ruby

```ruby
5 | 6 # gives 7

# At the bit level:
#     0101  (5)
#   | 0110  (6)
#   = 0111  (7)
```

XOR,

The **XOR** bitwise operation (or **exclusive or**) takes two sets of bits, and for each pair (the two bits at the same index in each bit set) returns 1 only if **one but not both** of the bits is 1. Otherwise, it returns 0.

```
1 ^ 1  →  0
1 ^ 0  →  1
0 ^ 1  →  1
0 ^ 0  →  0
```

Think of it like a bag of chips where only one hand can fit in at a time. If no one reaches for chips, no one gets chips, and if both people reach for chips, they can't fit and no one gets chips either!

When performing XOR on two integers, only digit columns used by **one but not both** integers remain:

Ruby

```ruby
5 ^ 6 # gives 3

# At the bit level:
#     0101  (5)
#   ^ 0110  (6)
#   = 0011  (3)
```

NOT

The **NOT** bitwise operation takes one set of bits, and for each bit returns 0 if the bit is 1, and 1 if the bit is 0.

```
~ 0  →  -1
~ 1  →  -2
```

When performing NOT on an integer, each bit of the integer is inverted.

Ruby

```ruby
~ 5 # gives -6

# At the bit level:
#   ~ 0000 0101  (5)
#   = 1111 1010  (-6)
```

If you're unsure why the resulting number is negative in this example, it's because numbers are represented using two's complement. Read up on binary numbers here (/concept/binary-numbers).

and bit shifts.

A **bit shift** moves each digit in a set of bits left or right. The last bit in the direction of the shift is lost, and a 0 bit is inserted on the other end.

```
0010 << 1  →  0100
1011 >> 1  →  0101
```

Bit shifts take number of times to shift as the right argument:

```
1010110 << 2  →  1011000
1011010 >> 3  →  0001011
```

A single left shift multiplies a binary number by 2:

```
0010 << 1  →  0100

0010 is 2
0100 is 4
```

Two left shifts multiplies by 4. Three left shifts multiplies by 8.

And similarly, shifting right *divides* by 2, throwing out any remainders.

Is one of those operations helpful for finding our unique integer?

We're seeing every integer twice, except one. Is there a bitwise operation that would let **the second occurrence of an integer cancel out the first?**

If so, we could start with a variable `unique_delivery_id` set to `0` and run some bitwise operation with that variable and each number in our array. If duplicate integers cancel each other out, then we'd only be left with the unique integer at the end!

Which bitwise operation would let us do that?

Solution

We XOR⅂ all the integers in the array. We start with a variable `unique_delivery_id` set to `0`. Every time we XOR with a new ID, it will change the bits. When we XOR with the same ID again, it will cancel out the earlier change.

In the end, we'll be left with the ID that appeared once!

Ruby

```ruby
def find_unique_delivery_id(delivery_ids)

    unique_delivery_id = 0

    delivery_ids.each do |delivery_id|
        unique_delivery_id ^= delivery_id
    end

    return unique_delivery_id
end
```

Complexity

$O(n)$ time, and $O(1)$ space.

What We Learned

This problem is a useful reminder of the power we can unlock by knowing what's happening at the "bit level." (/concept/binary-numbers)

How do you know when bit manipulation might be the key to solving a problem? Here are some signs to watch out for:

1. You want to multiply or divide by 2 (use a left shift (/concept/bit-shift) to multiply by 2, right shift (/concept/bit-shift) to divide by 2).
2. You want to "cancel out" matching numbers (use XOR (/concept/xor)).