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 a vector, `deliveryIdConfirmations`. When the drone comes back and lands, the ID is again added to the same vector.

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 vector 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 vector, we can always find the unique ID in **$O(1)$ 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 vector and **store each integer in an unordered map**, 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.

C++

```cpp
int findUniqueDeliveryId(const vector<int>& deliveryIds)
{
    unordered_map<int, int> idsToOccurrences;
    int uniqueId = 0;

    for (int deliveryId : deliveryIds) {
        auto it = idsToOccurrences.find(deliveryId);
        if (it != idsToOccurrences.end()) {
            ++it->second;
        }
        else {
            idsToOccurrences.insert(make_pair(deliveryId, 1));
        }
    }

    for (const auto& entry : idsToOccurrences) {
        if (entry.second == 1) {
            uniqueId = entry.first;
        }
    }

    return uniqueId;
}
```

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 unordered map. Can we bring that down?

Well, we could use booleans⤵ as our values, instead of integers. If we see an integer, we'll add it as a key in our unordered map 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⤵ using bits⤵ . What if we thought about this problem on the level of individual bits?

Let's think about the **bitwise operations** AND⤵ , OR⤵ , XOR⤵ , NOT⤵ and bit shifts⤵ .

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 `uniqueDeliveryId` set to `0` and run some bitwise operation with that variable and each number in our vector. 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 vector. We start with a variable `uniqueDeliveryId` 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!

C++

```cpp
int findUniqueDeliveryId(const vector<int>& deliveryIds)
{
    int uniqueDeliveryId = 0;

    for (int deliveryId : deliveryIds) {
        uniqueDeliveryId ^= deliveryId;
    }

    return uniqueDeliveryId;
}
```

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)).