Find a duplicate, *Space Edition*™.

We have an array of integers, where:

1. The integers are in the range $1..n$
2. The array has a length of $n + 1$

It follows that our array has *at least* one integer which appears *at least* twice. But it may have *several* duplicates, and each duplicate may appear *more than* twice.

Write a function which finds an integer that appears more than once in our array. (If there are multiple duplicates, you only need to find one of them.)

We're going to run this function on our new, super-hip Macbook Pro With Retina Display™. Thing is, the damn thing came with the RAM soldered right to the motherboard, so we can't upgrade our RAM. **So we need to optimize for space!**

Gotchas

We can do this in $O(1)$ space.

We can do this in less than $O(n^2)$ time, while keeping $O(1)$ space.

We can do this in $O(n \lg n)$ time and $O(1)$ space.

We can do this *without* destroying the input.

Breakdown

This one's a classic! We just do one walk through the array, using a set to keep track of which items we've seen!

Ruby

```ruby
require 'set'

def find_repeat(numbers)
    numbers_seen = Set.new
    numbers.each do |number|
        if numbers_seen.include? number
            return number
        else
            numbers_seen.add(number)
        end
    end

    # whoops--no duplicate
    raise Exception, 'no duplicate!'
end
```

Bam. $O(n)$ time and ... $O(n)$ space ...

Right, we're supposed to optimize for *space*. $O(n)$ is actually kinda high space-wise. Hm. We can probably get $O(1)$...

We can "brute force" this by taking each number in the range $1..n$ and, for each, walking through the array to see if it appears twice.

```ruby
def find_repeat_brute_force(numbers)
    (1...numbers.length).each do |needle|
        has_been_seen = false
        numbers.each do |number|
            if number == needle
                if has_been_seen
                    return number
                else
                    has_been_seen = true
                end
            end
        end
    end

    # whoops--no duplicate
    raise Exception, 'no duplicate!'
end
```

This is $O(1)$ space and $O(n^2)$ time.

That space complexity can't be beat, but the time cost seems a bit high. Can we do better?

One way to beat $O(n^2)$ time is to get $O(n \lg n)$ time. Sorting takes $O(n \lg n)$ time. And if we sorted the array, any duplicates would be right next to each-other!

But if we start off by sorting our array we'll need to take $O(n)$ space to store the sorted array...

...unless we sort the input array *in place*!

Okay, so this'll work:

1. Do an in-place sort of the array (for example an in-place mergesort).
2. Walk through the now-sorted array from left to right.
3. Return as soon as we find two adjacent numbers which are the same.

This'll keep us at $O(1)$ space and bring us down to $O(n \lg n)$ time.

But destroying the input is kind of a drag—it might cause problems elsewhere in our code. Can we maintain this time and space cost without destroying the input?

Let's take a step back. **How can we break this problem down into subproblems?**

If we're going to do $O(n \lg n)$ time, we'll probably be iteratively doubling something or iteratively cutting something in half. That's how we usually get a lgn. So what if we could cut the problem in half somehow?

Well binary search works by cutting the problem in half after figuring out which half of our input array holds the answer.

But in a binary search the *reason* we can confidently say which half has the answer is because the array is *sorted*. For this problem, when we cut our unsorted array in half we can't really make any strong statements about which elements are in the left half and which are in the right half.

What if we could cut the problem in half a *different* way, other than cutting the *array* in half?

With this problem, we're looking for a needle (a repeated number) in a haystack (array). What if instead of cutting the haystack in half, we cut *the set of possibilities for the needle* in half?

The full range of possibilites for our needle is $1..n$. How could we test whether the actual needle is in the first half of that range ($1..\frac{n}{2}$) or the second half ($\frac{n}{2}+1..n$)?

A quick note about how we're defining our ranges: when we take $\frac{n}{2}$ we're doing *integer division*, so we throw away the remainder. To see what's going on, we should look at what happens when n is even and when n is odd:

- If n is 6 (an even number), we have $\frac{n}{2} = 3$ and $\frac{n}{2} + 1 = 4$, so our ranges are $1..3$ and $4..6$.
- If n is 5 (an odd number), $\frac{n}{2} = 2$ (we throw out the remainder) and $\frac{n}{2} + 1 = 3$, so our ranges are $1..2$ and $3..5$.

So we can notice a few properties about our ranges:

1. They aren't necessarily the same size.
2. They don't overlap.
3. Taken *together*, they represent the original input array's range of $1..n$. In math terminology, we could say their *union* is $1..n$.

So, how do we know if the needle is in $1..\frac{n}{2}$ or $\frac{n}{2} + 1..n$?

Think about the original problem statement. We know that we have at least one repeat because there are $n + 1$ items and they are all in the range $1..n$, which contains only n distinct integers.

This notion of "we have more items than we have possibilities, so we must have at least one repeat" is pretty powerful. It's sometimes called the pigeonhole principle↴ . Can we exploit the pigeonhole principle to see which half of our range contains a repeat?

Imagine that we separated the input array into two subarrays—one containing the items in the range $1..\frac{n}{2}$ and the other containing the items in the range $\frac{n}{2} + 1..n$.

Each subarray has *a number of elements* as well as *a number of possible distinct integers* (that is, the length of the range of possible integers it holds).

Given what we know about the number of elements vs the number of possible distinct integers in the *original input array*, what can we say about the number of elements vs the number of distinct possible integers in *these subarrays*?

The sum of the subarrays' numbers of elements is $n + 1$ (the number of elements in the original input array) and the sum of the subarrays' numbers of possible distinct integers is n (the number of possible distinct integers in the original input array).

Since the sums of the subarrays' numbers of elements must be 1 greater than the sum of the subarrays' numbers of possible distinct integers, one of the subarrays must have at least one more element than it has possible distinct integers.

Not convinced? We can prove this by contradiction. Suppose neither array had more elements than it had possible distinct integers. In other words, both arrays have *at most* the same number of items as they have distinct possibilites. The sum of their numbers of items would then be *at most* the total number of possibilities across each of them, which is n. This is a contradiction—we know that our total number of items from the original input array is $n + 1$, which is greater than n.

Now that we know *one* of our subarrays has 1 or more items more than it has distinct possibilities, we know *that subarray* must have at least one duplicate, by the same pigeonhole argument that we use to know that the *original input aray* has at least one duplicate.

So once we know *which* subarray has the count higher than its number of distinct possibilites, we can use this same approach recursively, cutting *that* subarray into two halves, etc, until we have just 1 item left in our range.

Of course, we don't need to actually separate our array into subarrays. All we care about is *how long* each subarray would be. So we can simply do one walk through the input array, counting the number of items that *would be* in each subarray.

Can you formalize this in code?

Careful—if we do this recursively, we'll incur a space cost in the call stack! Do it iteratively instead.

Solution

Our approach is similar to a binary search, except we divide the *range of possible answers* in half at each step, rather than dividing the *array* in half.

1. Find the number of integers in our input array which lie within the range $1..\frac{n}{2}$.
2. Compare that to the number of possible unique integers in the same range.
3. If the number of *actual* integers *is greater* than the number of *possible* integers, we know there's a duplicate in the range $1..\frac{n}{2}$, so we iteratively use the same approach on that range.
4. If the number of actual integers *is not greater* than the number of possible integers, we know there must be duplicate in the range $\frac{n}{2} + 1..n$, so we iteratively use the same approach on that range.
5. At some point our range will contain just 1 integer, which will be our answer.

Ruby

```ruby
def find_repeat(the_array)
    floor = 1
    ceiling = the_array.length - 1

    while floor < ceiling

        # divide our range 1..n into an upper range and lower range
        # (such that they don't overlap)
        # lower range is floor..midpoint
        # upper range is midpoint+1..ceiling
        midpoint = floor + ((ceiling - floor) / 2)
        lower_range_floor, lower_range_ceiling = floor, midpoint
        upper_range_floor, upper_range_ceiling = midpoint+1, ceiling

        # count number of items in lower range
        items_in_lower_range = 0
        the_array.each do |item|
            # is it in the lower range?
            if item >= lower_range_floor and item <= lower_range_ceiling
                items_in_lower_range += 1
            end
        end

        distinct_possible_integers_in_lower_range = \
            lower_range_ceiling - lower_range_floor + 1

        if items_in_lower_range > distinct_possible_integers_in_lower_range
            # there must be a duplicate in the lower range
            # so use the same approach iteratively on that range
            floor, ceiling = lower_range_floor, lower_range_ceiling
        else
            # there must be a duplicate in the upper range
            # so use the same approach iteratively on that range
            floor, ceiling = upper_range_floor, upper_range_ceiling
        end
    end

    # floor and ceiling have converged
    # we found a number that repeats!
    return floor
```

```
end
```

Complexity

$O(1)$ space and $O(n \lg n)$ time.

Tricky as this solution is, we can actually do even better, getting our runtime down to $O(n)$ while keeping our space cost at $O(1)$. The solution is NUTS; it's probably outside the scope of what most interviewers would expect. But for the curious...here it is (/question/find-duplicate-optimize-for-space-beast-mode)!

Bonus

This function always returns *one* duplicate, but there may be several duplicates. Write a function that returns *all* duplicates.

What We Learned

Our answer was a modified binary search. We got there by *reasoning about the expected runtime*:

1. We started with an $O(n^2)$ "brute force" solution and wondered if we could do better.
2. We knew to beat $O(n^2)$ we'd probably do $O(n)$ or $O(n \lg n)$, so we started thinking of ways we might get an $O(n \lg n)$ runtime.
3. $\lg(n)$ usually comes from iteratively cutting stuff in half, so we arrived at the final algorithm by exploring that idea.

Starting with a target runtime and working *backwards* from there can be a powerful strategy for all kinds of coding interview questions.