Interview Cake

# Find a duplicate, *Space Edition*™.

We have a list of integers, where:

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

It follows that our list 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 list.** (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.

Most $O(n \lg n)$ algorithms double something or cut something in half. How can we rule out half of the numbers each time we iterate through the list?

## Breakdown

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

Python 3.6

```
def find_repeat(numbers):
    numbers_seen = set()
    for number in numbers:
        if number in numbers_seen:
            return number
        else:
            numbers_seen.add(number)

    # Whoops--no duplicate
    raise Exception('no duplicate!')
```

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 list to see if it appears twice.

Python 3.6

```
def find_repeat_brute_force(numbers):
    for needle in range(1, len(numbers)):
        has_been_seen = False
        for number in numbers:
            if number == needle:
                if has_been_seen:
                    return number
                else:
                    has_been_seen = True

    # Whoops--no duplicate
    raise Exception('no duplicate!')
```

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 list, any duplicates would be right next to each-other!

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

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

An **in-place** algorithm operates *directly* on its input and *changes* it, instead of creating and returning a *new* object. This is sometimes called **destructive**, since the original input is "destroyed" when it's edited to create the new output.

**Careful: "In-place" does *not* mean "without creating any additional variables!"** Rather, it means "without creating a new copy of the input." In general, an in-place function will only create additional variables that are $O(1)$ space.

Here are two functions that do the same operation, except one is in-place and the other is out-of-place:

Python 3.6

```
def square_list_in_place(int_list):
    for index, element in enumerate(int_list):
        int_list[index] *= element

    # NOTE: We could make this function just return, since
    # we modify int_list in place.
    return int_list


def square_list_out_of_place(int_list):
    # We allocate a new list with the length of the input list
    squared_list = [None] * len(int_list)

    for index, element in enumerate(int_list):
        squared_list[index] = element ** 2

    return squared_list
```

**Working in-place is a good way to save space.** An in-place algorithm will generally have $O(1)$ space cost.

**But be careful: an in-place algorithm can cause side effects.** Your input is "destroyed" or "altered," which can affect code *outside* of your function. For example:

```
original_list = [2, 3, 4, 5]
squared_list  = square_list_in_place(original_list)

print("squared: %s" % squared_list)
# Prints: squared: [4, 9, 16, 25]

print("original list: %s" % original_list)
# Prints: original list: [4, 9, 16, 25], confusingly!

# And if square_list_in_place() didn't return anything,
# which it could reasonably do, squared_list would be None!
```

Python 3.6

Generally, out-of-place algorithms are considered safer because they avoid side effects. You should only use an in-place algorithm if you're very space constrained or you're *positive* you don't need the original input anymore, even for debugging.

Okay, so this'll work:

1. Do an in-place sort of the list (for example an in-place merge sort).
2. Walk through the now-sorted list 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 "$\lg n$". So what if we could cut the problem in half somehow?

Well, binary search

**A binary search algorithm finds an item in a *sorted* list in $O(lg(n))$ time.**

A brute force search would walk through the whole list, taking $O(n)$ time in the worst case.

Let's say we have a sorted list of numbers. To find a number with a binary search, we:

1. **Start with the middle number: is it bigger or smaller than our target number?** Since the list is sorted, this tells us if the target would be in the *left* half or the *right* half of our list.
2. **We've effectively divided the problem in half**. We can "rule out" the whole half of the list that we know doesn't contain the target number.
3. **Repeat the same approach (of starting in the middle) on the new half-size problem**. Then do it again and again, until we either find the number or "rule out" the whole set.

We can do this recursively, or iteratively. Here's an iterative version:

Python 3.6

```
def binary_search(target, nums):
    """See if target appears in nums"""

    # We think of floor_index and ceiling_index as "walls" around
    # the possible positions of our target, so by -1 below we mean
    # to start our wall "to the left" of the 0th index
    # (we *don't* mean "the last index")
    floor_index = -1
    ceiling_index = len(nums)

    # If there isn't at least 1 index between floor and ceiling,
    # we've run out of guesses and the number must not be present
    while floor_index + 1 < ceiling_index:

        # Find the index ~halfway between the floor and ceiling
        # We use integer division, so we'll never get a "half index"
        distance = ceiling_index - floor_index
        half_distance = distance // 2
        guess_index = floor_index + half_distance

        guess_value = nums[guess_index]

        if guess_value == target:
            return True

        if guess_value > target:
            # Target is to the left, so move ceiling to the left
            ceiling_index = guess_index
        else:
            # Target is to the right, so move floor to the right
            floor_index = guess_index

    return False
```

**How did we know the time cost of binary search was $O(lg(n))$?** The only non-constant part of our time cost is the number of times our while loop runs. Each step of our while loop cuts the range (dictated by `floor_index` and `ceiling_index`) in half, until our range has just one element left.

**So the question is, "how many times must we divide our original list size ($n$) in half until we get down to 1?"**

$$n * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * ... = 1$$

How many $\frac{1}{2}$'s are there? We don't know yet, but we can call that number $x$:

$$n * (\frac{1}{2})^x = 1$$

Now we solve for $x$:

$$n * \frac{1^x}{2^x} = 1$$

$$n * \frac{1}{2^x} = 1$$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

Now to get the $x$ out of the exponent. How do we do that? Logarithms.

**Recall that $\log_{10} 100$ means, "what power must we raise 10 to, to get 100"? The answer is 2.**

So in this case, if we take the $\log_2$ of both sides...

$$\log_2 n = \log_2 2^x$$

The right hand side asks, "what power must we raise $2$ to, to get $2^x$?" Well, that's just $x$.

$$\log_2 n = x$$

So there it is. The number of times we must divide $n$ in half to get down to 1 is $log_2 n$. So our total time cost is $O(lg(n))$

**Careful: we can only use binary search if the input list is *already sorted*.**

works by cutting the problem in half after figuring out which half of our input list holds the answer.

But in a binary search, the *reason* we can confidently say which half has the answer is because the list is *sorted*. For this problem, when we cut our unsorted list 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 *list* in half?

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

The full range of possibilities 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 list'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.

> The **pigeonhole principle** states that if $n$ items are put into $m$ containers, with $n > m$, then at least one container must contain more than one item.
>
> For example, there must be at least two left gloves or two right gloves in a group of three gloves.

Can we exploit the pigeonhole principle to see which half of our range contains a repeat?

Imagine that we separated the input list into two sublists—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 sublist 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 list*, what can we say about the number of elements vs the number of distinct possible integers in *these sublists*?

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

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

Not convinced? We can prove this by contradiction. Suppose neither list had more elements than it had possible distinct integers. In other words, both lists have *at most* the same number of items as they have distinct possibilities. 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 list is $n+1$, which is greater than $n$.

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

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

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

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 *list* in half.

1. Find the number of integers in our input list 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.

Python 3.6

```
def find_repeat(the_list):
    floor = 1
    ceiling = len(the_list) - 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
        for item in the_list:
            # Is it in the lower range?
            if item >= lower_range_floor and item <= lower_range_ceiling:
                items_in_lower_range += 1

        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

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

## 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 *backward* from there can be a powerful strategy for all kinds of coding interview questions.

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.