I want to learn some big words so people think I'm smart.

I opened up a dictionary to a page in the middle and started flipping through, looking for words I didn't know. I put each word I didn't know at increasing indices in a huge array I created in memory. When I reached the end of the dictionary, I started from the beginning and did the same thing until I reached the page I started at.

Now I have an array of words that are mostly alphabetical, except they start somewhere in the middle of the alphabet, reach the end, and then start from the beginning of the alphabet. In other words, this is an alphabetically ordered array that has been "rotated." For example:

```ruby
words = [
    'ptolemaic',
    'retrograde',
    'supplant',
    'undulate',
    'xenoepist',
    'asymptote', # <-- rotates here!
    'babka',
    'banoffee',
    'engender',
    'karpatka',
    'othellolagkage',
]
```

Write a function for finding the index of the "rotation point," which is where I started working from the beginning of the dictionary. This array is huge (there are lots of words I don't know) so we want to be efficient here.

Gotchas

We can get $O(\lg n)$ time.

Breakdown

The array is *mostly* ordered. We should exploit that fact.

What's a common algorithm that takes advantage of the fact that an array is sorted to find an item efficiently?

Binary search!⤵

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

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

Let's say we have a sorted array 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 array is sorted, this tells us if the target would be in the *left* half or the *right* half of our array.
2. **We've effectively divided the problem in half**. We can "rule out" the whole half of the array 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:

Ruby ▾

```ruby
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 = nums.length

    # 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]

        return true if guess_value == target

        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
        end
    end

    return false
end
```

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 array size (n) in half until we get down to 1?"

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

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 array is *already sorted*.

We can write an adapted version of binary search for this.

In each iteration of our binary search,↴ how do we know if the rotation point is to our left or to our right?

Try drawing out an example array!

Ruby ▾

```
words = [ 'k','v','a','b','c','d','e','g','i' ]
                          ^
```

If our "current guess" is the middle item, which is 'c' in this case, is the rotation point to the left or to the right? How do we know?

Notice that every item to the *right* of our rotation point is always alphabetically *before* the first item in the array.

So the rotation point is to our *left* if the current item is less than the first item. Else it's to our right.

Solution

This is a modified version of binary search. At each iteration, we go right if the item we're looking at is greater than the first item and we go left if the item we're looking at is less than the first item.

We keep track of the lower and upper bounds on the rotation point, calling them `floor_index` and `ceiling_index` (initially we called them "floor" and "ceiling," but because we didn't imply the type in the name we got confused and created bugs). When `floor_index` and `ceiling_index` are directly next to each other, we know the floor is the last item we added before starting from the beginning of the dictionary, and the ceiling is the first item we added after.

```ruby
def find_rotation_point(words)

    first_word = words[0]

    floor_index = 0
    ceiling_index = words.length - 1

    while floor_index < ceiling_index

        # guess a point halfway between floor and ceiling
        guess_index = floor_index + ((ceiling_index - floor_index) / 2)

        # if guess comes after first word or is the first word
        if words[guess_index] >= first_word
            # go right
            floor_index = guess_index
        else
            # go left
            ceiling_index = guess_index
        end

        # if floor and ceiling have converged
        if floor_index + 1 == ceiling_index

            # between floor and ceiling is where we flipped to the beginning
            # so ceiling is alphabetically first
            return ceiling_index
        end
    end
end
```

Complexity

Each time we go through the while loop, we cut our range of indices in half, just like binary search. So we have $O(\lg n)$ loop iterations.

In each loop iteration, we do some arithmetic and a string comparison. The arithmetic is constant time, but the string comparison requires looking at characters in both words—*every* character in the worst case. Here, we'll assume our word lengths are bounded by some constant so we'll say the string comparison takes constant time.

> The longest word in English is *pneumonoultramicroscopicsilicovolcanoconiosis*, a medical term. It's 45 letters long.

Putting everything together, we do $O(\lg n)$ iterations, and each iteration is $O(1)$ time. So our time complexity is $O(\lg n)$.

> Some languages—like German, Russian, and Dutch—can have arbitrarily long words, so we might want to factor the length of the words into our runtime. We could say the length of the words is ℓ, each string comparison takes $O(\ell)$ time, and the whole algorithm takes $O(\ell * \lg n)$ time.

We use $O(1)$ space to store the first word and the floor and ceiling indices.

Bonus

This function *assumes* that the array is rotated. If it isn't, what index will it return? How can we fix our function to return 0 for an unrotated array?

What We Learned

The answer was a modified version of binary search.

This is a great example of the difference between "knowing" something and *knowing* something. You might have *seen* binary search before, but that doesn't help you much unless you've *learned the lessons of binary search*.

Binary search teaches us that *when an array is sorted or mostly sorted*:

1. The value at a given index tells us a lot about what's to the left and what's to the right.
2. We don't have to look at every item in the array. By inspecting the middle item, we can "rule out" *half* of the array.
3. We can use this approach over and over, cutting the problem in half until we have the answer. This is sometimes called "divide and conquer."

So whenever you know an array is sorted or almost sorted, think about these lessons from binary search and see if they apply.