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 list 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 a list 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 list that has been "rotated." For example:

```python
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 list 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 list is *mostly* ordered. We should exploit that fact.

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

Binary search! 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 list!

```python
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 list.

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.

Python

```
def find_rotation_point(words):
    first_word = words[0]
    floor_index = 0
    ceiling_index = len(words) - 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

        # 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
```

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 list is rotated. If it isn't, what index will it return? How can we fix our function to return 0 for an unrotated list?

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 a list 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 list. By inspecting the middle item, we can "rule out" *half* of the list.
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 a list is sorted or almost sorted, think about these lessons from binary search and see if they apply.