# 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 vector 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 vector 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 vector that has been "rotated." For example:

```cpp
const vector<string> 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 vector 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 set is *mostly* ordered. We should exploit that fact.

What's a common algorithm that takes advantage of the fact that a set 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 vector!

```cpp
[ "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 vector.

**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 `floorIndex` and `ceilingIndex` (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 `floorIndex` and `ceilingIndex` 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.

```cpp
size_t findRotationPoint(const vector<string>& words)
{
    size_t floorIndex = 0;
    size_t ceilingIndex = words.size() - 1;

    while (floorIndex < ceilingIndex) {

        // guess a point halfway between floor and ceiling
        size_t guessIndex = floorIndex + ((ceilingIndex - floorIndex) / 2);

        // if guess comes after first word or is the first word
        if (words[guessIndex] >= words[0]) {
            // go right
            floorIndex = guessIndex;
        }
        else {
            // go left
            ceilingIndex = guessIndex;
        }

        // if floor and ceiling have converged
        if (floorIndex + 1 == ceilingIndex) {

            // between floor and ceiling is where we flipped to the beginning
            // so ceiling is alphabetically first
            break;
        }
    }

    return ceilingIndex;
}
```

# Complexity

$O(\lg n)$ time and $O(1)$ space, just like binary search.

We're assuming that our word lengths are bound by some constant—if they were bounded by a non-constant $l$, each of our string comparisons would cost $O(l)$, for a total of $O(l * \lg n)$ runtime.

# Bonus

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

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

Want more coding interview help?

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