Interview Cake

Suppose we had an array↴ of n integers *sorted in ascending order*. How quickly could we check if a given integer is in the array?

Solution

Because the array is sorted, we can use 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:

JavaScript ▼

```
function binarySearch(target, nums) {
    // see if target appears in nums

    // we think of floorIndex and ceilingIndex 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")
    var floorIndex = -1;
    var ceilingIndex = 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 (floorIndex + 1 < ceilingIndex) {

        // find the index ~halfway between the floor and ceiling
        // we have to round down, to avoid getting a "half index"
        var distance = ceilingIndex - floorIndex;
        var halfDistance = Math.floor(distance / 2);
        var guessIndex = floorIndex + halfDistance;

        var guessValue = nums[guessIndex];

        if (guessValue === target) {
            return true;
        }

        if (guessValue > target) {

            // target is to the left, so move ceiling to the left
            ceilingIndex = guessIndex;

        } else {

            // target is to the right, so move floor to the right
            floorIndex = guessIndex;
        }
    }

    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 `floorIndex` and `ceilingIndex`) 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?

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

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*.

to find the item in $O(\lg n)$ time and $O(1)$ additional space.

Want more coding interview help?

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