

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

## Solution

Because the list is sorted, we can use 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:

```
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?"**

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

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

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

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*.**

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.