— Binary Search —

"I now took numbers behind the doors, but I sort them for you. So they're not in the same random order like they were for Naomira. You now have the advantage to know that the numbers are sorted from small to big.

Where might you propose we begin the story this time? With which locker?

Let's find number 6 this time."

Still being open sort of an artifact of the greater efficiency, it would seem, of this algorithm because now that Rave was given the assumption that these numbers are sorted from small on the left to large on the right, she was able to apply that same divide and conquer algorithm from week zero which we're now going to give a name — binary search.

And simply by starting in the middle and realizing, "ok, too small", then by going to the right half and realizing, "oh, went a little too far, then by going to the left half, which, Rave able to find in just 3 steps instead of seven the number six in this case that we were actually searching for.

So you can see that this would seem to be more efficient.

So if I had used different numbers but still sorted them from ~~right~~ left to right, would it still have worked this algorithm?

So long as the numbers are always in the same order from left to right or they could even be in reverse order, so long as it's consistent, the decisions that Rave was making — if greater than, else, if less than — would guide us to the solution no matter what.

And it would seem to take FEWER steps. So if we consider now the pseudocode for this algorithm, let's take a look how we might describe binary search.

So binary search we might describe with something like this:

If number behind middle door
    Return true
Else if the number is less than the middle door,
    Search left half
Else if number > middle door
    Search right half
Return false

But if there's no doors — and we'll see in a moment WHY I put this up top just to keep things clean.

↓

More similar with C:

↓

If no doors
    Return false
} safety condition

If number behind doors [middle] → the middle door
    Return true
Else if number < doors [middle]
    Search doors [0] through doors [middle - 1]
Else if number > doors [middle]
    Search doors [middle + 1] through doors [n - 1]

"search the left half, search the right half — but start to now describe it in terms of actual indices or indexes like we did with our array notation.

The last scenario is if the number is greater than the door's bracket middle, then Rave would have wanted to search the middle door + 1 (so 1 over) through doors n-1

So again, just a way of sort of describing a little more syntatically what it is that's going on.

So how might we translate this now into big O notation? Well, in the worst case, how many steps total might Rave's binary search algorithm have taken?

Given 8 doors or given more generically n doors, how many times could she go left or go right before finding herself with one or no doors left? $\boxed{\log n}$

So there's log n again

And even if you're not feeling wholly comfortable with your logarithm still, pretty much in programming and in computer science more generally, any time we talk about some algorithm that's dividing and conquering in half or any other multiple, it's possibly involving logarithms in some sense.

And log n essentially refers to the number of times you can divide n by 2 until you bottom out at just a single door or equivalently zero doors left.

So we might say that indeed binary search is in big O of log n → $\boxed{O(\log n)}$

Because the door that Rave opened last, this one, happened to be three doors away. And actually, if you do the math here, that roughly works out to be exactly that case.

If we add one, that's sort of out of seven doors or roughly eight, we were able to search it in just 3 total steps.

What about omega notation, though?

In the best case, Rave might have gotten lucky. She opened the door, and there it is...
So how might we describe a lower bound on the running time of binary search?

$\Omega(1)$

So here too, we see that in some cases, binary search and linear search, they're pretty equivalent and so this is why sometimes compelling to consider both the best case in the worst case because honestly, in general, who really cares if you just get lucky once in a while and your algorithm is super fast?

What you probably care about is what's the worst case? How long am I going to be sitting there watching some spinning hourglass or beach ball trying to give myself an answer to a pretty big problem?

Well, odds are, you're going to generally care about big O notation. So indeed, moving forward, we'll generally talk about the running time of algorithms often in terms of big O, a little less so in terms of omega.

But understanding the range can be important depending on the nature of the data that you're going to actually be given here.

SORTING AND Searching
X
JUST SEARCHING

"So this method is clearly more efficient, but it requires that the information is all compiled in a certain order. How do you ensure that you can compile information in a particular order at scale?"

"If I can generalize it, how do you guarantee that you can do this at scale, which algorithm is better?"

I've sort of led us down this road of implying that Dave's second algorithm, binary search, is better because it's so much faster.

It's log n in the worst case instead of big O of n.

But Dave was given an advantage when she came up here in that the doors were already sorted. And so that sort of invites the question, given a whole bunch of random data, either a small data set or, heck, something Google sized with millions, billions pieces of data, should you sort it first from smallest to largest and then search? Or should you first dive right in and search it linearly?

How might you think about that?

If you're google, should they first go with linear search because it's always going to work even though it might be slow? Or should they invest the time in sorting all of that data — we'll see how in a bit — and then search it more efficiently?

If you had to sort the data first and we don't yet formally know how to do this, but obviously, as humans, we could probably figure it out.

You do have to look at all of the data anyway.

And so you're sort of wasting your time if you're sorting it only then to go in search it.

But maybe it depends a bit more... Like, that's absolutely right, and if you're just searching for 1 thing in life, then that's probably a waste of time to sort it and then search it because you're just adding to the process.

But what's another scenario in which you might not worry about that whereby it might make sense to sort it and then search?

So if your problem is a Google-like problem where you have more than just 1 user who's searching for more than just one website page, probably you should incur the cost up front and sort the whole thing because every subsequent request thereafter is going to be faster, faster and faster because it's going to (?) algorithm of binary search, that's going to add up to be way fewer steps than doing linear search multiple times.

Kind of depends on the use case and kind of depends on how important it is. And this happens even in real world contexts.

"What is your most precious resource?"
↳ is it time to run the code?
↳ to write the code?
↳ amount of memory the computer is using?
it really depends on what your goals are. ☺