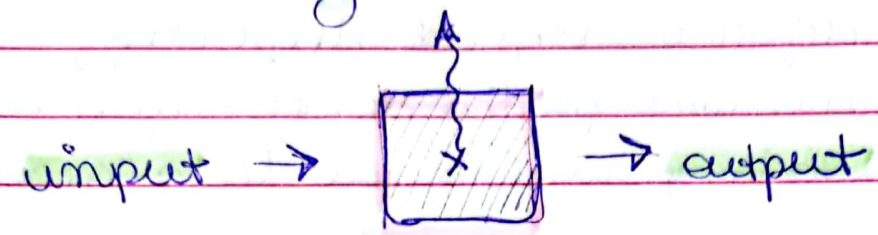
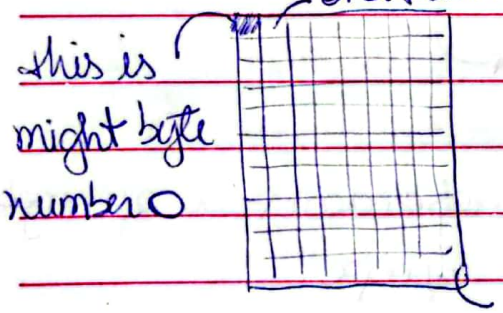
Week 3

Algorithms



R.A.M → Random Access Memory

We looked at one of these little black chips that contains all of the bytes — all of the bits, ultimately. It's just kind of a grid sort of an artist grid, that allows you to think about every one of these memory locations as just having a number or an address, so to speak.



And so as we did that, we started to explore how we could use this canvas to create kind of our information, our own maybe down here something like 2 billion if you have 2 GB of memory.

inputs and outputs, not just the basics like ints and floats and so forth, but we also talked about STRINGS.

And what is a string as you now know it?

~ An array of characters ~

How might someone else define an array in more familiar

now turns? What would be an array?
Kind of like an indexed set of things. (?)
A key characteristic to keep in mind with an array is that it does actually pertain to memory. And it's contiguous memory. → Byte after byte is what constitutes an array.

And we'll see in a couple of weeks time that there's actually new interesting ways to use this same primitive known to stitch together things that are sort of 2 directional, even that have some kind of shape to them.
But for now, all we've talked about is arrays and just using these things from left to right, top to bottom, contiguously to represent information.

So today, we'll consider still an array, but we won't focus so much on representation of strings or other data types.

We'll actually NOW focus on the other part of that process, of inputs becoming outputs, namely the thing in the middle — ALGORITHMS.

But we have to keep in mind, even though every time we've looked at an array thus far, certainly on the ~~board~~ board like this, you as a human certainly have the luxury of just kind of eyeballing the whole

thing with a bird's eye view and seeing where all of those numbers are.

If I asked you where a particular number is, like zero, odds are your eyes would go RIGHT to where it is, and boom, problem solved in sort of one step.

But the catch is, with a computer that has this memory, even though you, the human can see everything at once, a computer cannot. It's better to think of your computer's memory, your phone's memory, or more specifically an array of memory like this as really being a set of closed doors, not unlike lockers in a school.

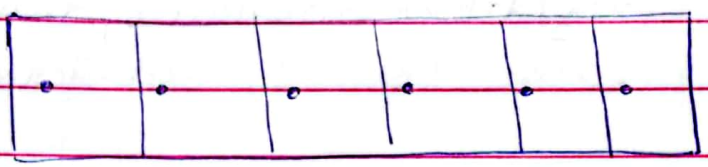
And only by opening each of those doors can the computer actually see what's in there, which is to say that the computer, unlike you, doesn't have this bird's eye view of all of the data in all these locations.

It has to much more methodically look here, maybe look here, and so forth in order to find something.

Now fortunately, we already have some building blocks — loops, conditions, boolean expressions, and the like where you could imagine writing some code that very methodically goes from LEFT to RIGHT or RIGHT to LEFT or something more sophisticated

that actually finds something you're looking for.

And just remember, that the convention we've had since last week now is that these arrays are ZERO indexed, so to speak. To be ZERO indexed just means that the data type starts counting from ZERO.



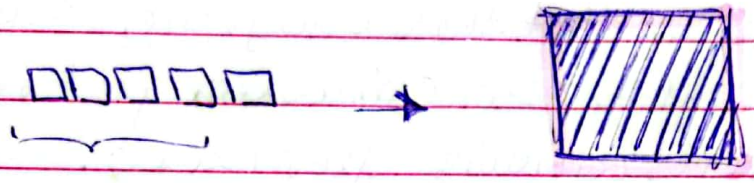
So in the general case, if you had (n) doors or (n) bytes of memory, 0 would always be at the LEFT, and n - 1 would always be at the RIGHT.

SEARCHING

→ What does it mean to SEARCH for something? to find information on this, of course, is omnipresent. Search is kind of one of the MOST important topics and features of any device these days.

↓ Let's consider HOW the Googles, the Apples, the Microsofts of the world are implementing something as seemingly familiar as this.

So here might be the problem statement: we want some INPUT to become some OUTPUT.



what's that input going to be?

to get back an answer, true or false?

→ Is something we're looking for there or not?

You can imagine taking this one step further and trying to find where is the thing you're looking for.

But for now, let's just take 1 bite out of the problem. Can we tell ourselves, true or false, is some number behind one of these doors or lockers in memory?

But before we go there and start talking about ways to do that - that is, ALGORITHMS.

Let's consider HOW we might lay the foundation of, like, comparing whether one algorithm is better than another.

We talked about correctness, and it sort of goes without saying that ANY code you write, any algorithm you implement, should had better be correct.

Otherwise, what's the point if it doesn't give you the right answers? ☺

But we also talked about design. And in your own words, what do we mean when we say a program is better DESIGNED at this stage than another?

How do you think about this notion of design now?
1 — easier to understand;
2 — Efficiency (doesn't use up too much memory);
3 — Quality of the code but also the quality of the performance;

And as our programs get bigger and more sophisticated and just longer, those kinds of things are really going to matter.

And in a real world, if you start writing code not just by yourself but with someone else, getting the design right is just going to make it easier to collaborate and ultimately produce write code, with just higher probability.

So let's consider how we might focus on exactly the second characteristic, the efficiency, of an algorithm, and the way we might talk about the efficiency of algorithms, just how fast or how slow they are, is in terms of their running time.

That is to say, when they're running, how much time do they take? And we might measure this in seconds or milliseconds or minutes or just some number of steps in the general case because presumably fewer steps, to

your point, is better than more steps;

So how might we think about running times?

↳ Well, there's one general notation we should define today. So computer scientists tend to describe the running time of an algorithm or a piece of code, for that matter, in terms of what's called "big O notation."