# Computational Thinking

This is CS50. Harvard University. Fall 2015.

Cheng Gong

## Table of Contents



# Introduction

- CS50 for MBAs[1] is an introductory course to computer science, based on the introductory computer science course at Harvard College.
- While the undergraduate course is very hands-on, this course is more top-down and intended to help technical and non-technical folks make engineering decisions or have technical conversations.
- The technologies and topics we want to try to cover include:
  # Computational Thinking
    # The "essence" of computer science, if you will.
  # Programming
    # We'll cover a graphical language called Scratch, which has the basic constructs that most other languages use.
  # Algorithms, Data Structures
    # Solving problems in computer science often reduce to the same basic ways and structures, so we'll look at some of those.
  # Privacy, Security
    # We'll include some current events like the current Apple vs. FBI case, in looking at the implications of prviacy and security of technology in society.

[1] https://cs50.harvard.edu/mba/

    # Internet
        # We'll look at how the Internet actually works.
    # Databases
        # We'll look at how lots of data might be stored efficiently and quickly.
    # Web Development
        # What it means to actually create a website, with languages like HTML and CSS.
    # Web Programming
        # Building a website more programmatically, with languages like PHP or Ruby or Python, so the website can have dynamic, changing content.
    # Technology Stacks
        # Includes buzzwords like frameworks and libraries, but choosing technology stacks can mean making important decisions when solving some technical problem.
    # Mobile
        # The implications of making websites and apps for mobile devices.
- The course will be more instruction-oriented, though we will talk about current events and startups.
- There will be assignments to help reinforce materials from the previous class and to prepare for the next class.
- There will be three projects that are more programming-heavy, but will allow for a real taste of what programming is actually like.

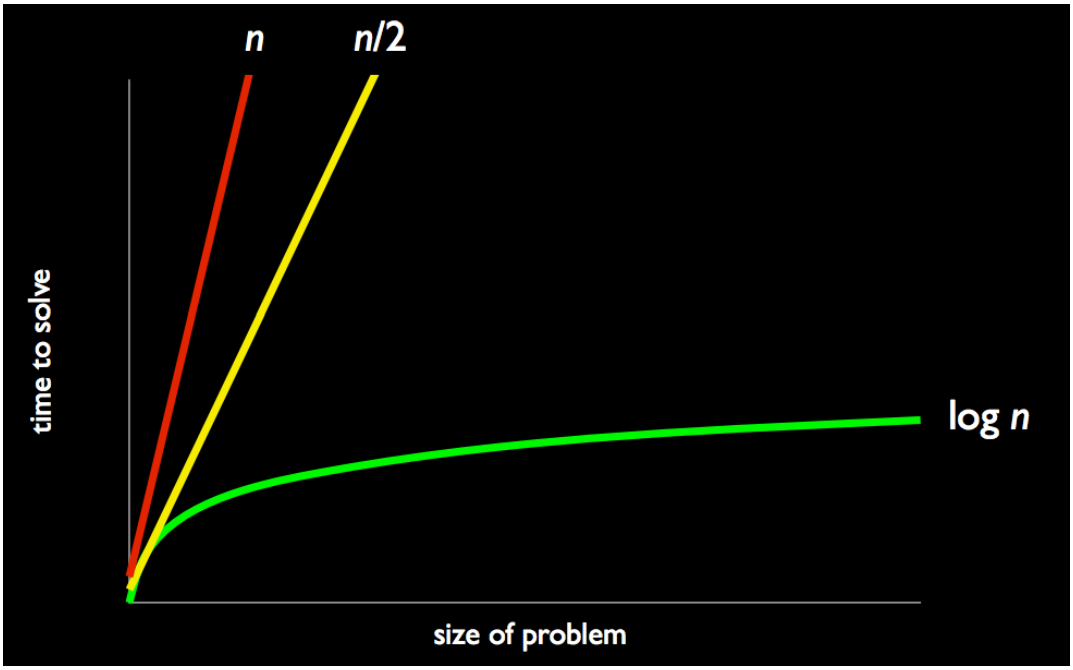
## Computational Thinking

- We'll look at a few higher-level principles of "thinking like a computer":
    # ASCII, binary
        # How data is stored in a computer.
    # abstraction, algorithms, pseudocode
    # imprecision, overflow
        # Some problems that can arise.

- Let’s propose that, at the most basic level, computational thinking involves three things:
  # inputs (what we have or know before we solve problems)
  # algorithm (some procedure for solving problems)
  # outputs (what we hope to have after solving problems)
- So first we need to represent inputs and outputs somehow.
- The most basic form of physical input to a computer is electricity, whether something is on or off. Using those two states, we can actually represent all the information we want, with patterns of on and off, or 0 and 1. For example, turning on the light on a phone might represent 1, and turning it off might be a 0.
- But with more phones, we can count up higher. For example, turning on one of two phones might represent 1, and turning on the other might be 2, and turning on both might represent 3. With a third phone, we can count even higher.
- This system of using two digits is called **binary**. Humans normally use **decimal**, or 10 digits, 0-9, to count. It turns out that decimal is quite similar to binary, or any other mode of counting.
- Let’s take a look at `123`, which we would normally read as "a hundred and twenty-three." Well, each number is in a column, or a place, with the 1s place on the right, the 10s place in the middle, and the 100s place on the left. So we have `1` times 100, `2` times 10, and `3` times 1, for a hundred and twenty-three altogether.
- We can extend that to binary, where the columns will represent, from right to left, 1s, 2s, and 4s. In the previous example with decimal, the 1s place was 10^0, the 10s place was 10^1, and the 100s place was 10^2. So the places in binary are powers of 2, where the rightmost column is 2^0, the middle is 2^1, and the right is 2^2. (And this can continue indefinitely, just like it can in decimal!)
- So we can represent the number 1 as `001` in binary, 2 as `010` in binary, 3 as `011`, and 4 as `100`. Notice that we might need to change more than one digit sometimes, as we do when counting from, say, 99 to 100 in decimal.
- To get 8, we’d need another digit, since with three bits `111` is equivalent to 7. Adding 1 to that, we’d get `1000`, which has four bits.
- Notice that adding a 1 led to us carrying the extra digit over. In the human world, we take this ability for granted, since we can just write the extra digit somewhere to the left. But in computers, we have only a finite number of bits, whether they are on a hard drive or in other kinds of memory.

- So this is a limitation of computers. If we had only three phones, the biggest number we could represent is 7, and if we tried to represent 8, we wouldn't be able to, unless we had another phone. In computers, adding two very large numbers might result in **overflow**, where each number might fit into some number of bits, but their sum might be too large, and the extra information would be lost. With three phones representing 7, adding 1 would result in the appearance of 0, since all three of them would be off.
- Since computers have so much memory nowadays, we simplify by saying that one byte is 8 bits, and a thousand bytes is a kilobyte, a million bytes a megabyte, a billion a gigabyte, and a trillion bytes a terabyte.
- We can also represent decimal numbers in another system, where some bits of 0s and 1s represent the number to the left of the decimal point, and some bits represent the number to the right of the decimal point.
- With 3 bits, we can represent 8 numbers, between 0 and 7 inclusive. Generally, with a fixed number of bits, we can only represent a fixed number of, erm, numbers. But there are an infinite number of real numbers, so at some point a computer loses some precision in storing a decimal number.
- We can ask a computer to print out 1 divided by 10, which we would think is 0.1, and it does print `0.10000000` when we ask it for 8 digits. But if we ask it for 20 digits, we get `0.10000000000000000555`. With 30 digits, we get `0.100000000000000005551115123126`. Perhaps we might be looking too far in memory and looking at bits belonging to another program, which could happen, but is not the case here.
- The answer is that this number is represented with 32 bits, which (doing some math), we know can only represent about 4 billion distinct values. Because of this, the computer is showing us the closest match it can with that number of bits. (Imagine trying to write down the value of 1/3 with only 5 decimal digits. The best we could do is 0.33333. Similarly, a computer can only store 1/10 to the accurace of the digits we see above.)
- And even though we've long known this, there are lots of implications where computers used to control things, especially militarily, can lead to serious consequences due to imprecision.
- So we'll now take for granted that computers can represent information with 0s and 1s.
- Now let's move on to the last component we need, **algorithms**. Algorithms are just a set of instructions for solving some problem.

- Let’s take a phone book, with, say, 1000 pages. And say that one of the pages contain the name Mike Smith.
- We could open the book and flip from the beginning, one page at a time, until we reached Mike Smith. And this algorithm would be correct, since we’d eventually reach his name. But this isn’t a particular efficient, or fast, solution.
- We could go twice as fast by flipping two pages at a time, but this algorithm would no longer be correct since we might skip Mike Smith if we got unlucky and he was on a page we skipped. Though we can fix this by going back one page if we get to "Sn" instead of "Sm" and haven’t found Mike.
- But let’s do something simple, like opening the book to the middle. We’ll see that we’re in the Ms, which means that Mike Smith must be in the second half of the phone book, as S comes after M. So we can tear this problem in half, since we know we can get rid of the first half. We flip to the middle of what we have left, see that we’re in the Ts, and keep just the left half since S comes before T. And we’ll repeat this until we have one page, hopefully with Mike Smith’s name on it.
- A computer scientist would describe an algorithm like this as recursive, or applying the same algorithm again and again until we have a solution.
- We can plot the relationship between the size of the problem and the time it takes to solve in this graph:



    # The red line, labeled `n`, is our first algorithm where we flipped one page at a time, meaning that a problem of size `n` would take `n` steps to solve. In our case, a phone book with 1000 pages would require 1000 page flips, and every additional page would require an additional step.

    # The yellow line, labeled `n/2`, is our next algorithm where we flipped two pages at a time. This is slightly more efficient than the red line, but still straight as every additional two pages add one more step, regardless of the size of the problem.

    # Finally, the green line, labeled `log n`, is saying that the time required to solve a problem grows logarithmically. In our case, with our halving algorithm, even a phone book twice the size would only take one additional step.

- A computer scientist would say that the first algorithm is on the order of n steps, O(*n*). The second algorithm is O(*n/2*), but we can simplify it to be O(*n*) since the constant factor doesn't affect how the algorithm grows that much. And finally the last algorithm is O(*log(n)*), which means it grows much more slowly.
- We chose some familiar problems to show that computer science is closer to problem solving that you might think, and that we want to find algorithms that are both correct and efficient.

## Pseudocode

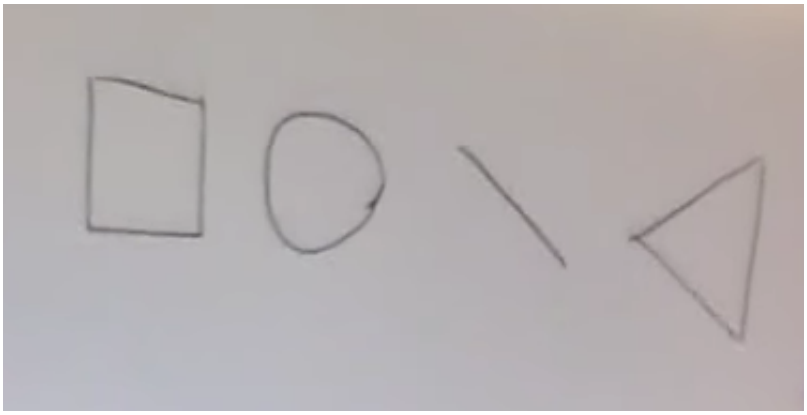
- Let's use **pseudocode**, something English-like, to explain our algorithm:

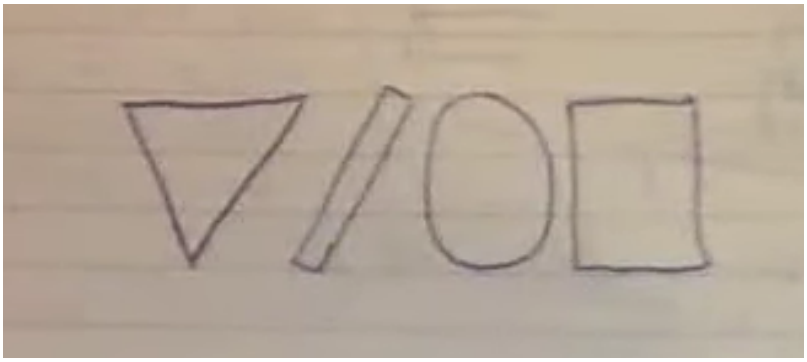
```
open phone book to middle
look for person you're looking for
if person you're looking for < current page
    divide book in half, throw away right half, go to (2)
else if person you're looking for > current page
    divide book in half, throw away left half, go to (2)
else if person is on current page
    call them
else
    give up
```

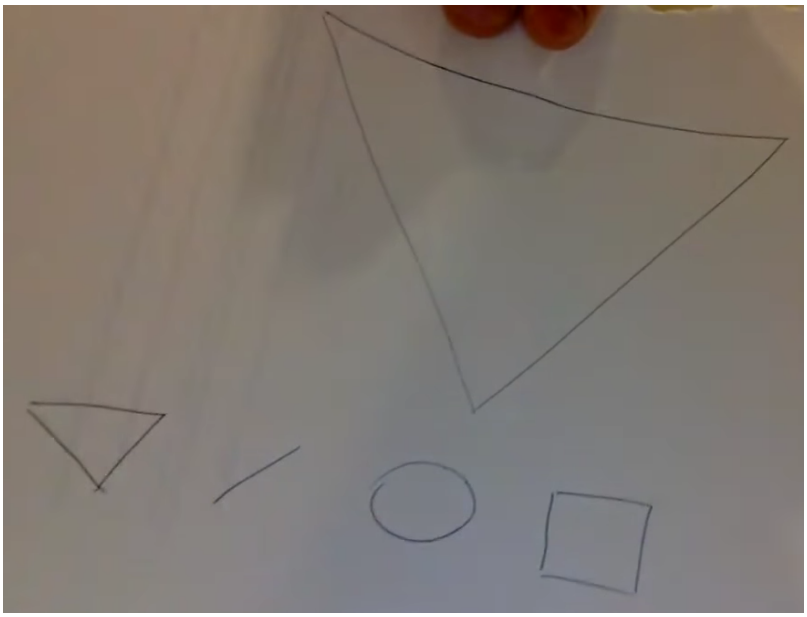
    # This is an example of iteration, or looping, since we're repeating things over and over again. Line 1, for example, is called a statement, whereby we are doing something.

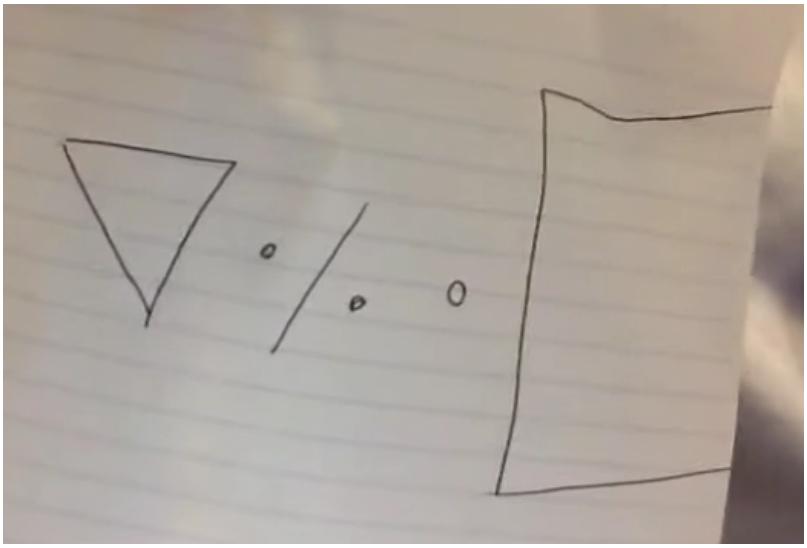
# Line 3 is a condition, where we decide to do a certain action (in this case line 4) if the condition is true. The indenting helps us visually see that line 4 will only be done if line 3 applies to the situation. The same idea applies to lines 5 and 6, 7 and 8, and 9 and 10.

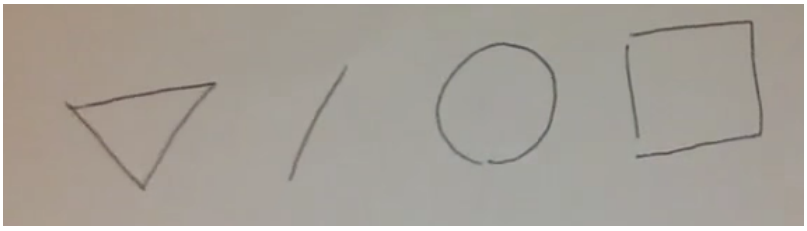
- There is also an idea of abstraction, or by looking at problems at a relatively high level. For example, line 2, "look for person you're looking for", sounds intuitive. But it is an abstraction, since a computer (or a very small child) might not be able to understand that, and that too needs to be defined somehow. Perhaps "look for person you're looking for" means that look at each of the names on a page, and compare it to the name we want. And we can go even further and ask what it means to read words on page, and suddenly line 2 can be 5 lines or even more of statements. So abstractions help us solve problems at a higher level.
- Another case in point, Uber's app has a map built in, and that map looks very much like Google's maps. And while they aren't affiliated with each other, Uber is able to build on top of Google's work, without having to create an entire set of maps themselves.
- Let's do a fun activity with the audience. A volunteer comes down to look at a drawing that only she can see, and instructs the audience how to recreate the drawing on their own sheets of paper.
- Her first instruction: "An upside-down triangle." Then, "there are four symbols inside." She continues with various instructions, and once she is done we compare some samples from the audience:



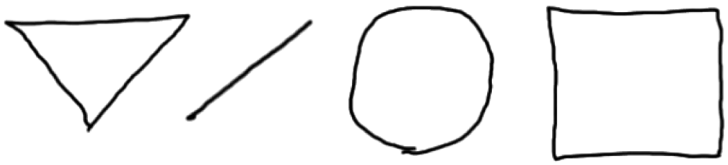




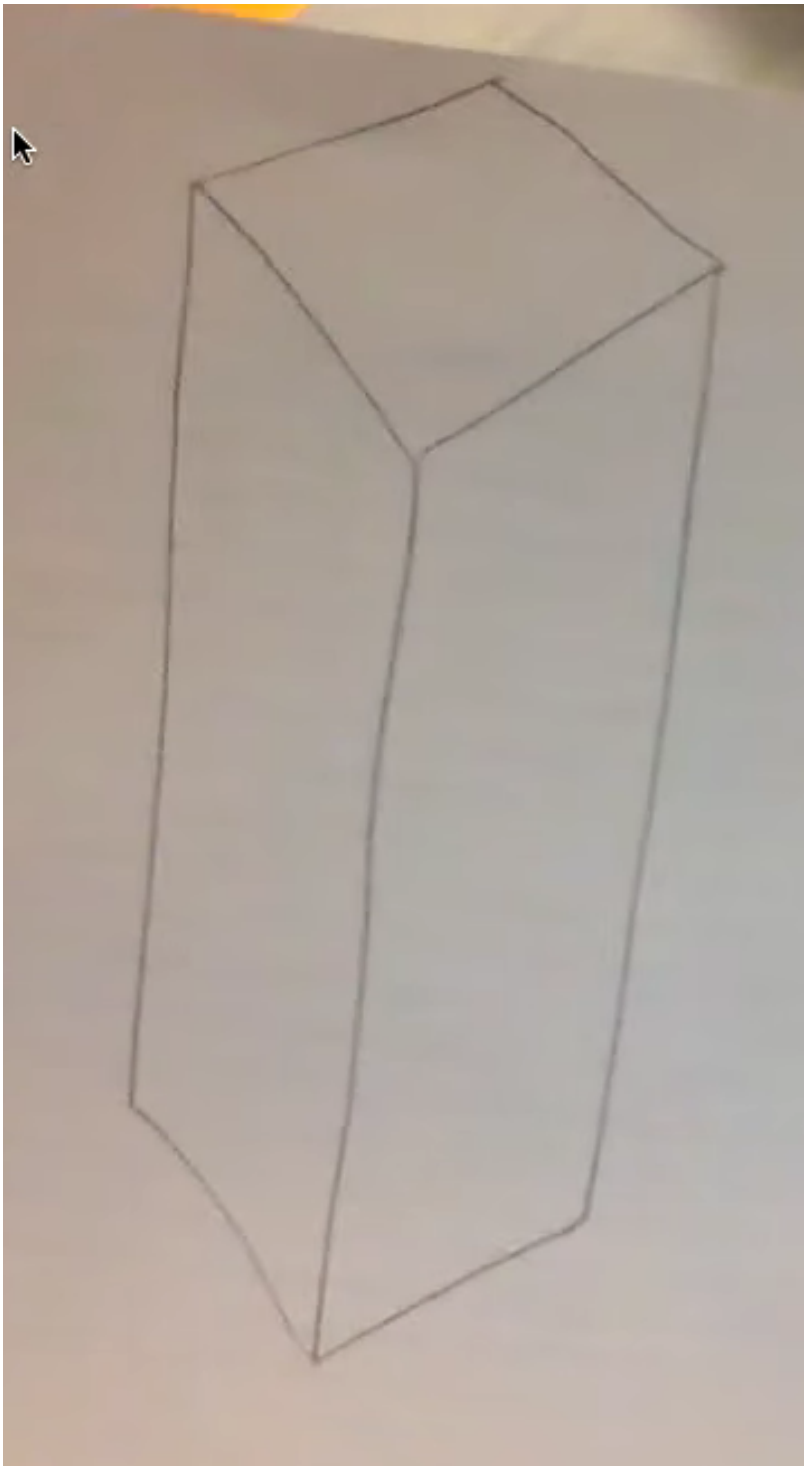


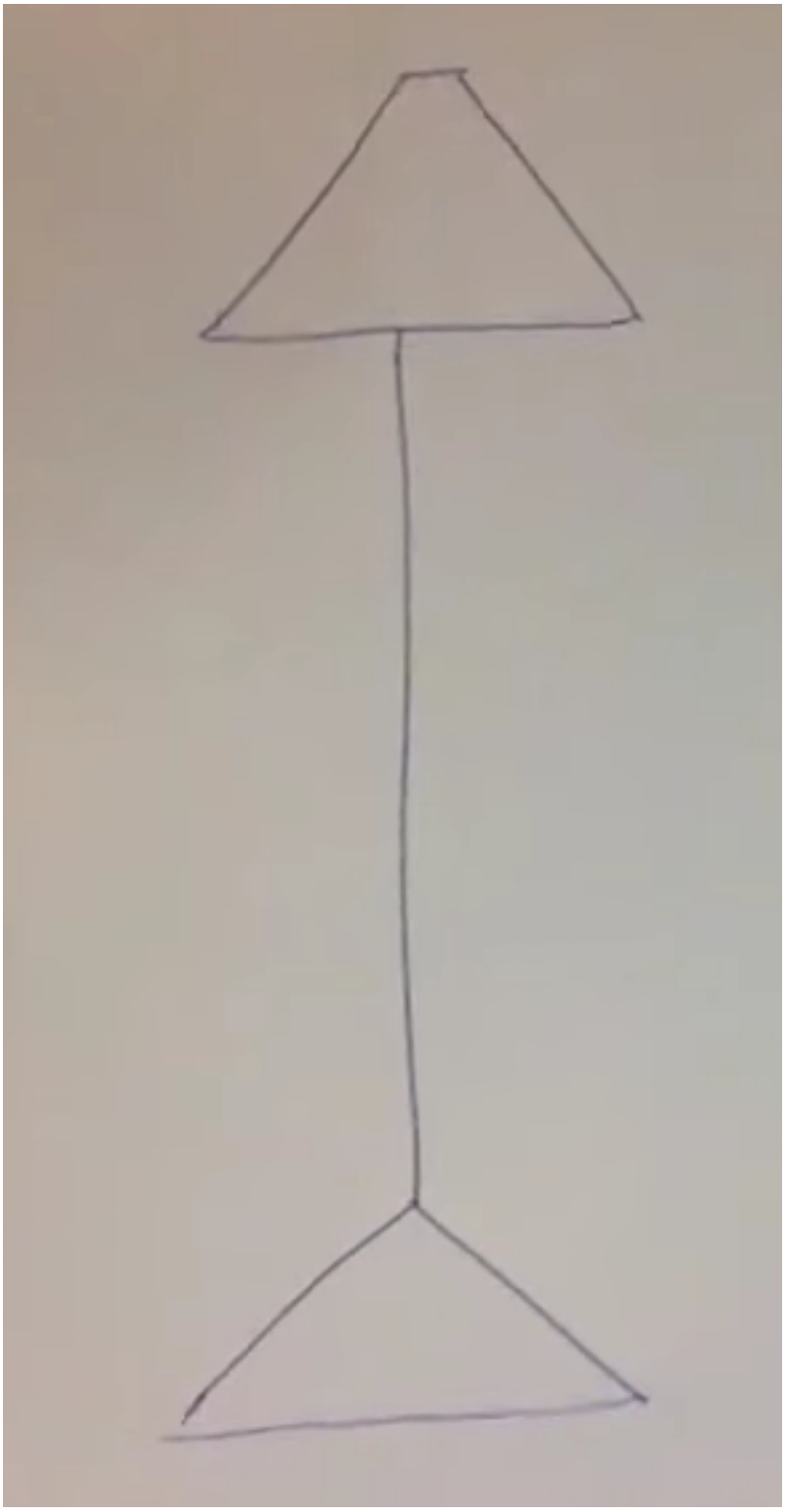


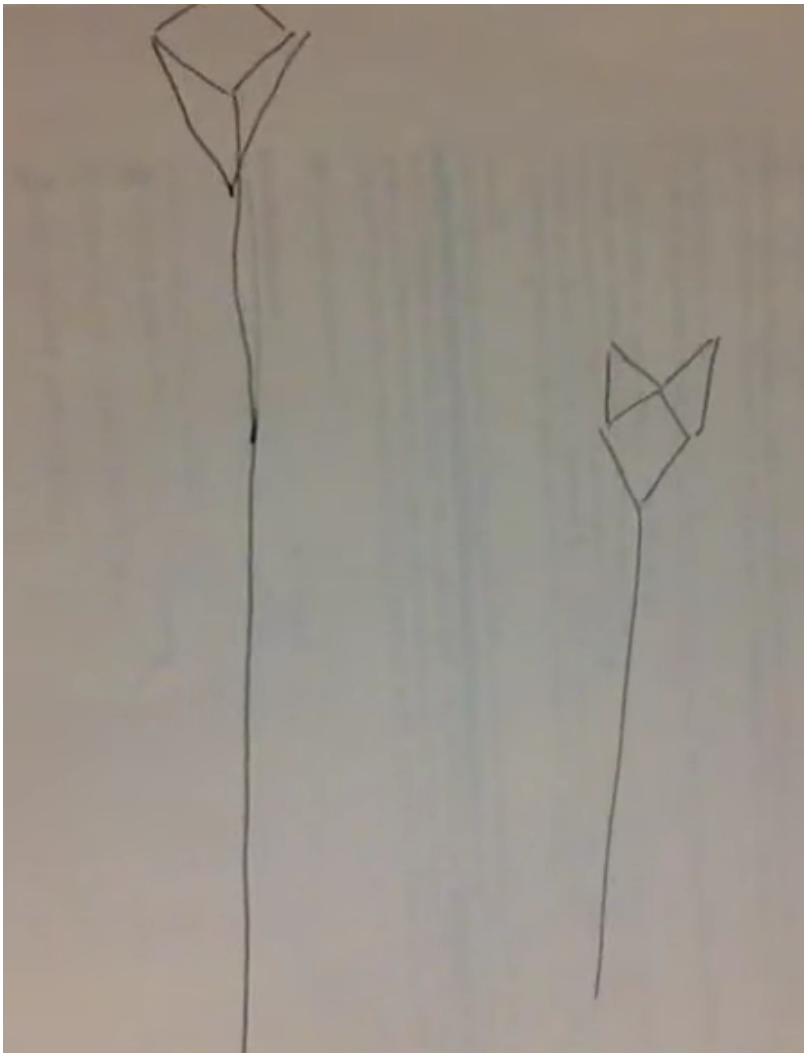
- And the original image looked like this:

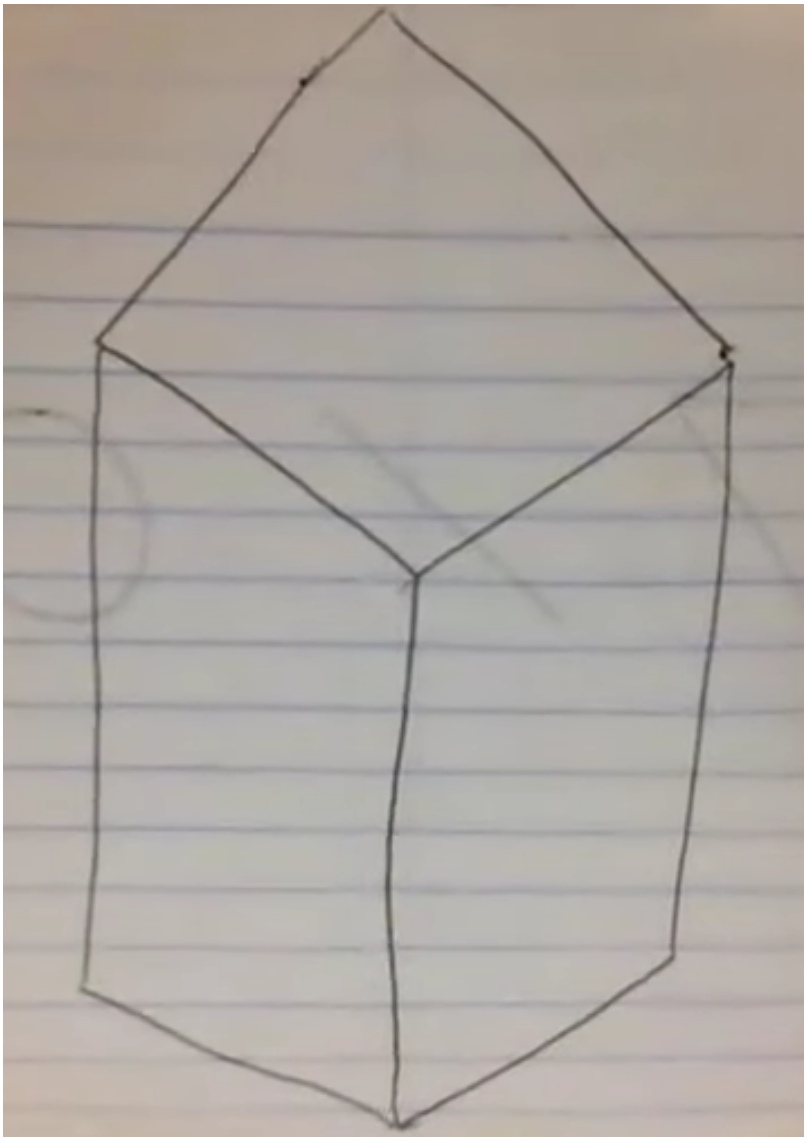


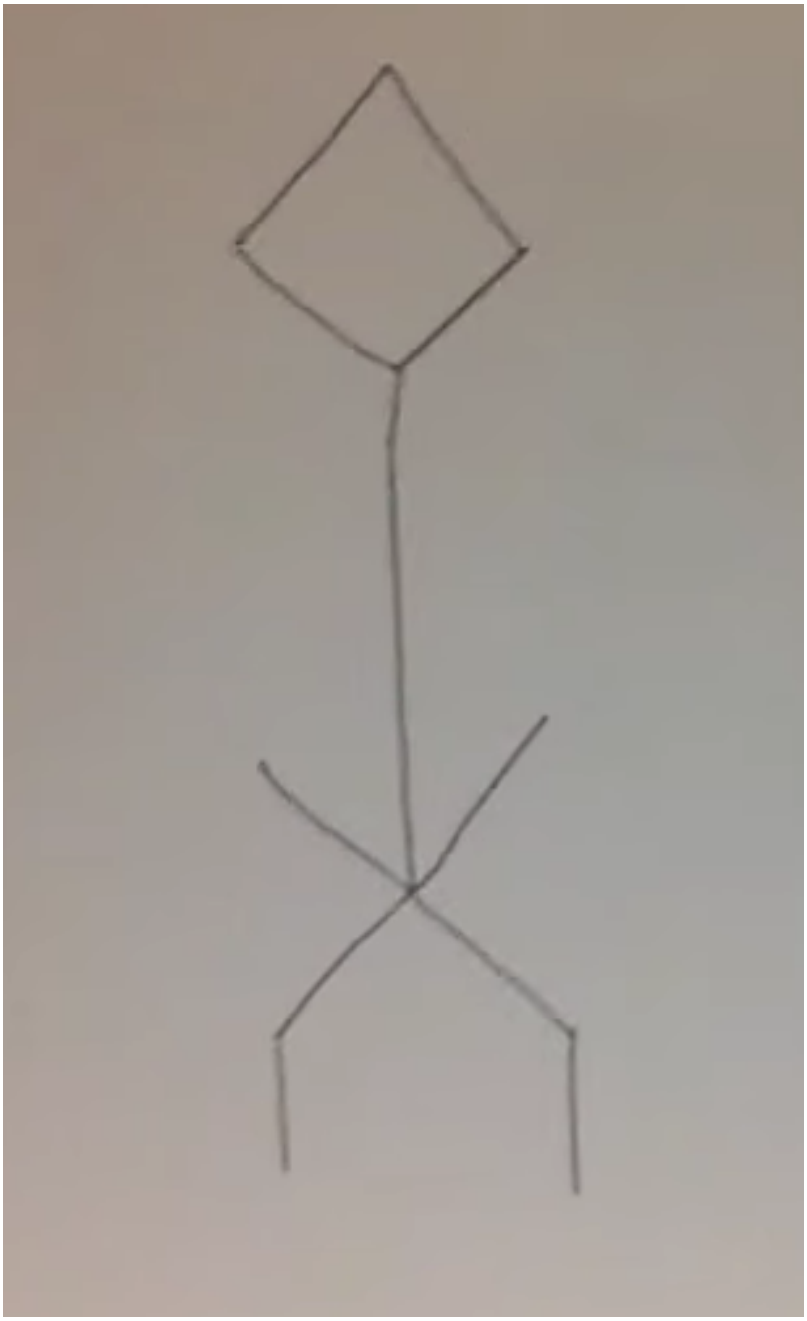
- In fact, some people were pretty close. But there were many opportunities to get lost, and many opportunities at which to make assumptions.
- We'll try this one more time with another volunteer, who starts with instructions like "draw a vertical line straight up in the middle of the paper" and "on top of that draw a diamond, like a lollipop with a diamond".
- Again we take some samples, which are more varied now:

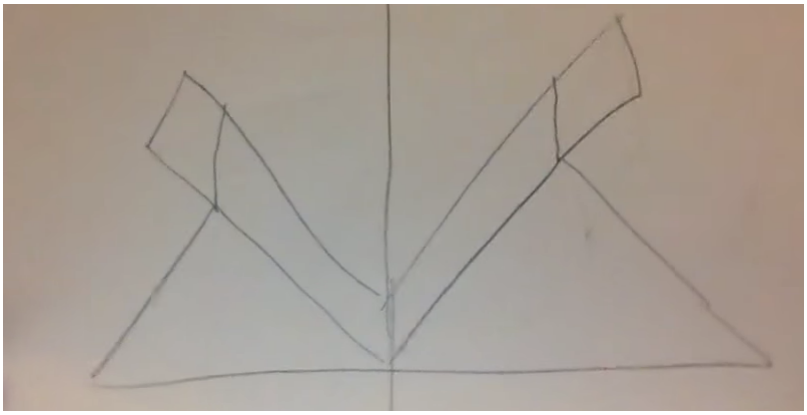




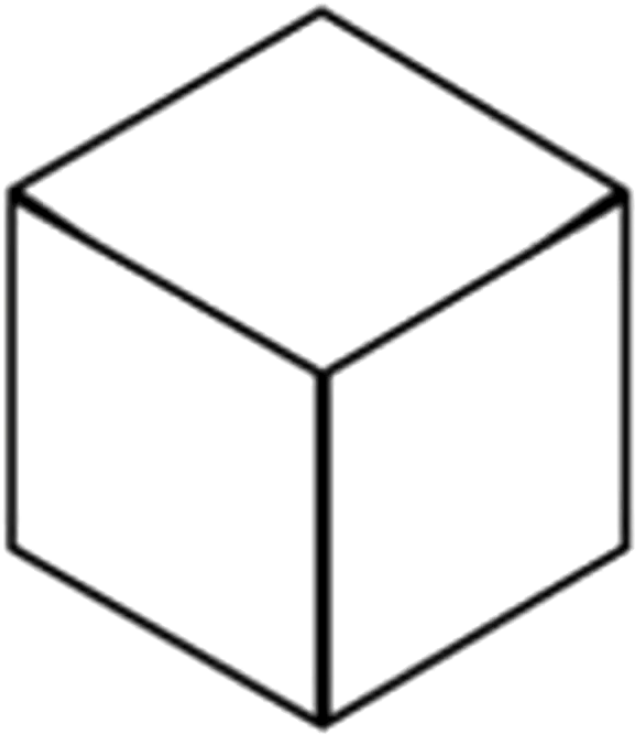




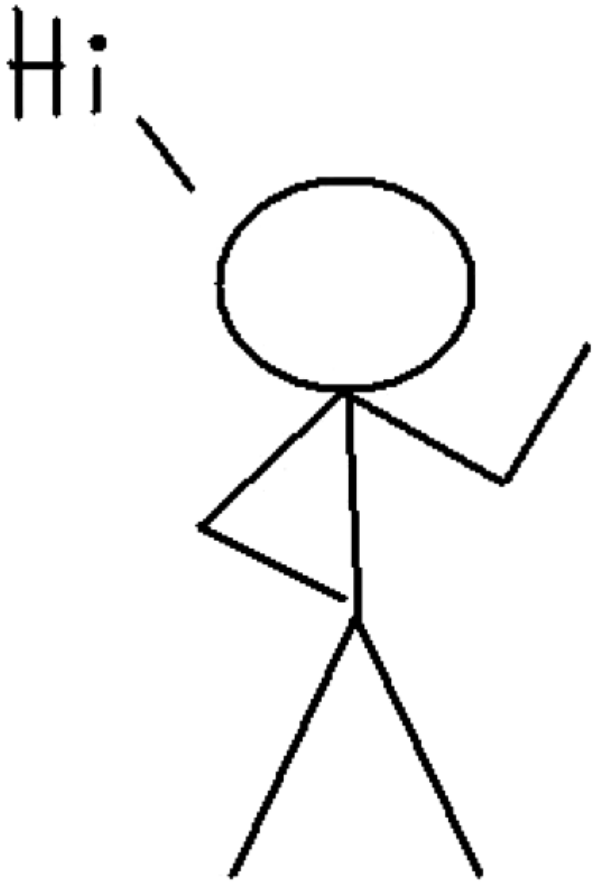




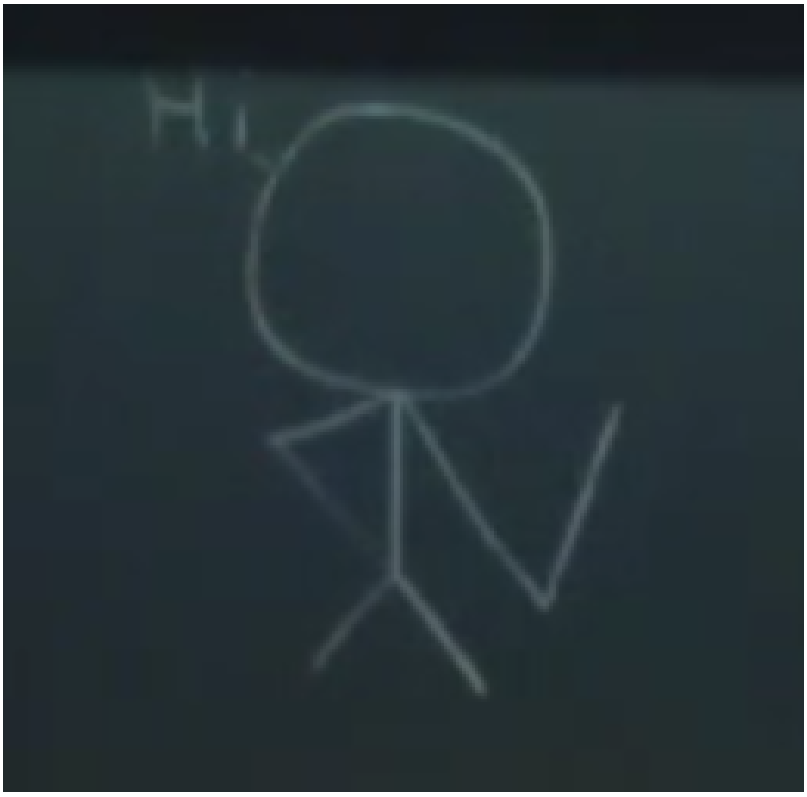
- The image was actually just a cube:



- Our volunteer could have just said "draw a cube", but instead he tried to instruct the audience to draw it one line at a time. But at one point he referred to drawing lines like a "Y", and that’s another example of abstraction, where he assumed we knew what that meant.
- Now we will have a volunteer try to draw this picture, with the audience providing instructions:




- The first instruction was to "draw a circle", and then "below that draw an upside-down Y". Then we struggled with describing a "triangle on the left of the line of the Y".
- We continue with specific instructions until we finally get:



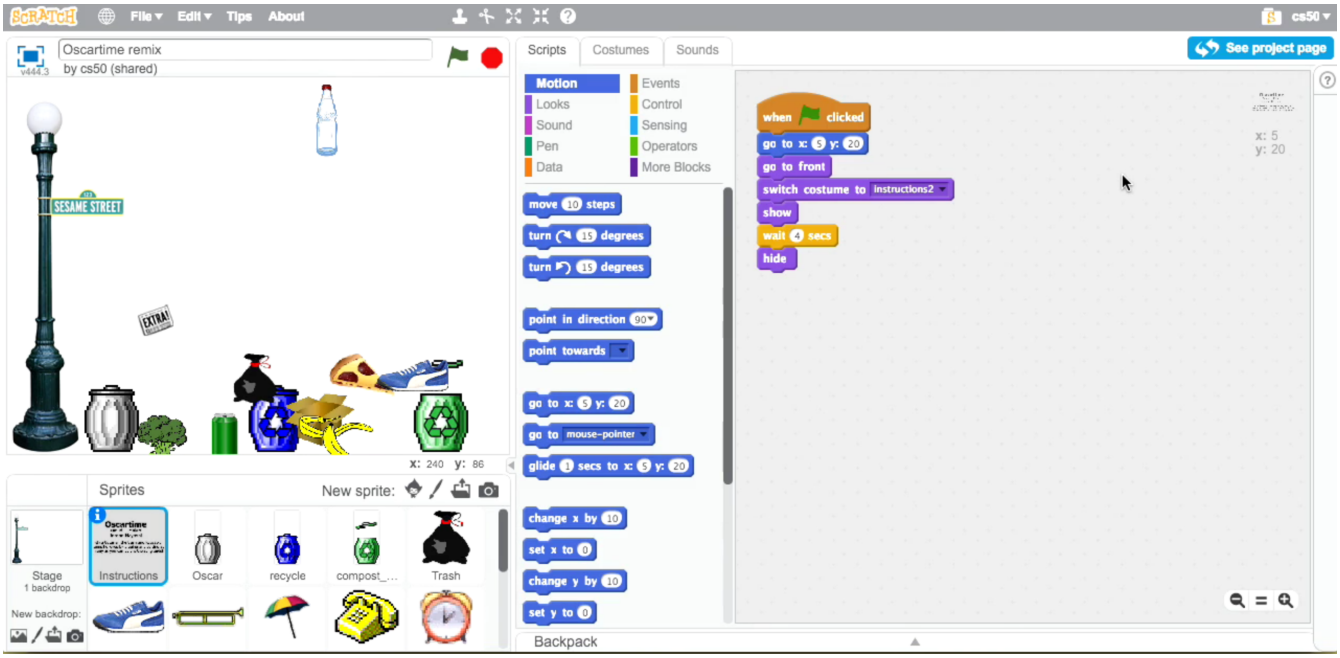
- So that was pretty close, and we can see how imprecision in our instructions might lead a computer to do something we don't expect. Perhaps there might be some case that a programmer did not expect, so there might be some endless cycle we are stuck in, or simply something not working as we'd like.
- Tomorrow we'll focus on programming, or translating these computational ideas to code with a graphical programming language called Scratch:

  # Boolean expressions, conditions, functions, loops, variables

  # events, threads

- Check out some examples like Oscartime remix[2]. We can imagine that this is built with basic constructs like items randomly starting at some horizontal position, and "falling" from the top by changing the vertical position in some loop.
- Tomorrow when we actually use Scratch you'll see a screen like this, where the right side is like a palette for our instructions that look like puzzle pieces:

[2] https://scratch.mit.edu/projects/76196420/




- And we'll take these small ideas and build on them to better understand bigger-picture topics. Until next time!