Day 2

This is CS50 for MBAs. Harvard Busines School. Spring 2015.

Cheng Gong

Table of Contents

Day 1 recap

- We have two objectives today: one is to get an idea of how to solve engineering problems in the real world, and one is to learn some basically programming concepts.
- Office hours[1], in addition to immediately after class, will be Thursday and Sunday afternoons.
- Question: When does the class end each day?
  - # We're scheduled for 3:15, but some days we might finish early, so we'll try to have that posted on the website in advance for people to plan.
- Question: Does the expression "hashtag" have anything to do with hashing tags so they can be looked up easily?
  - # The origin story is a nice read, but basically in 2007 someone tweeted about using the pound (also known as hash) symbol (#) to group related posts, and it's stuck since. It was inspired by IRC, Internet Relay Chat, which was around before Twitter, which also used the # symbol to refer to room, or channel, names.
- Remember that yesterday we discussed arrays, data structures where all the elements are of the same type and contiguous (laid out next to one another in memory). Because of this, we can access any element in constant time.

[1] http://cs50.net/mba/hours/

- A linked list took $O(n)$ time to access elements, since each element points to the next one which is might be stored far away, so we have to follow each arrow, or pointer.
- We claim that sorting takes a minimum of n steps, too, because we need to at least look at every element to know that it's in the right place.
- In general, we talk about algorithms in terms of O running time because we care more about the slowest it might be in the worst case, and not the fastest in the best case.
- If you're really interested in how the math for $O(n \log n)$ was derived for merge sort, you can look here.
- If both searching for elements in linked lists and hash tables with chaining take linear time, why is a hash table actually faster? Well, if we think back to our example from yesterday, searching $n/31$ elements is faster than searching through n elements, even if we might need to look through the entire list linearly in both cases. And we can also think of our hash table as an array vertically and a linked list horizontally.

  # In Python, hash tables are actually called dictionaries, and in JavaScript an object, so we'll start to see this theme of abstraction, where the programming for the lower-level details have already been done in so-called libraries that we can just use.

Dropbox

- Last night we watched a quick video on what algorithms were, and now we'll look at how they're applied in the real world.
- Dropbox[2] is a file synchronization service, copying files between computers for you automatically. You can also share files and folders publically, or specific people by giving them URLs.
- Friends at Dropbox say hi and tell us about their roles via video.
- In particular, we need to think about redundancy, or how we might keep data stored so we don't lose anything when a hard drive physically fails.
- One way to prevent this is to just copy data to multiple drives, but when you change some file you have to copy over the changes. The biggest cost, though, is the literal cost of the drives because now we need to purchase some multiple of the size we actually use, just to hedge against the risk of one of them failing.

[2] http://dropbox.com/

- Another feature of Dropbox is its version control, or how it keeps track of these changes over time. As humans, we could manually save files with names like `resume-1` or `resume-2` as we update them, but that's really annoying to do.
- But going back to data redundancy, Dropbox has thousands and thousands of users, and if they only stored each user's data on one drive across hundreds or thousands of drives, the chance of one of them failing and some user losing data is actually quite high.
- Even if the data is replicated, Dropbox has to make sure that it's stored on different physical drives, on different machines, on different racks, and even in different rows within a data center, since there could be some failure that impacts an entire row of machines. On top of that, geographical redundancy is necessary too, in the event of, say, an earthquake or internet outage. The cost of this becomes quite expensive, since we need so many copies.
- But there's a more creative way to offer the same amount of redundancy without maintaining so many copies. Erasure coding is a mathematical technique that takes, say, 6 separate files, and combines them into 9 separate blocks. The storage overhead is now 9/6, or 1.5x, but we can lose any three of those blocks, and still be able to reconstruct all 6 files. In particular, the erasure coding algorithm used is called Reed-Solomon[3].
- Dropbox actually uses a higher ratio than 1.5x, since they want to maintain a higher level of redundancy, to handle almost any unexpected incident.
- There are tradeoffs to using erasure coding, too: disks might need to move around more to gather the information for a file, and there are computational costs involved when storing or decoding a file.
- Let's say that there are two phrases we want to store, `foo` and `bar`. (These silly words are just conventionally used by computer scientist as stand-ins for real strings or names.)
- We don't want to store each string twice, but maybe add another string such that we can recover the other two strings, even if we lost one of them.
- Compression is one idea, but we could be doing that in addition to trying to solve the redundancy issue, since it saves space.

[3] http://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction

- Remember that the computer represents things in binary. Let's say we had two super-short files which are just `101` and `111` in binary. We could add a third number that reads `010`.
- We made this number by following the rule that the bit in the third number will be `0` if the bits in the same column in the other two are the same, and a `1` if the bits in the same column in the other two are different. The first column of the third number, given the other two, for example, is `0` since there is a `1` in the first column of the first number and a `1` in the first column of the second number.
- Now that we've done this, if we lose any number, we can recreate it by following the same rule based on the other numbers we have left. If we lost the second number, for example, we can see that the first bit would be a `1` because the first column of the first number is `1` and the first column of the third number is `0`, and so on.
- This isn't quite a complicated algorithm as erasure coding, but instead relies on an operation called **XOR**, exclusive or. That just means exactly what we did with each column. If two bits are the same, then the result will be `0`, but if they are different ("exclusively different") then the result will be `1`.

  # RAID[4] uses this property, but with hard drives. For example, if you had two drives with data, you could have a third drive that contained the XOR of all the bits in the first and second drives, so if you lost any one of them, you could recreate the data.
- Fancier algorithms allow you to have not one but say *n* extra hard drives, such that if you lose any *n* of them, you can still recover all the data.
- Another assumption here is that all the hard drives are the same size, or at least limited to the size of the smallest one.
- Another consideration is deduplication, where if multiple users happen to upload the same file, the data isn't stored multiple times. (Of course, we still copy it in some form to provide redundancy, but we don't want to do that more than we need to.)
- One way to check if files are the same is to read both, and compare the bits. If we knew the file sizes, we could take a shortcut by comparing that first, and know the files were different right away if the sizes were different.
- We could also randomly sample bits within files and compare those, but then we have a small chance that two users have slightly different files that are mistaken for the same, so one of them ends up with a different one after.

[4] http://en.wikipedia.org/wiki/RAID

    # In fact, certain file formats, like Word documents, start with a certain number of bits that are always the same, to indicate the format they are.

- The way that Dropbox solves this problem is to take a hash of each file, and just compare those values.
- To get these hashes, we use hash function like SHA256. Remember our hash table from yesterday, where the hash function took a person as input and returned a value from 1 to 31 based on the day they were born. In computing, hash functions generally take in any number of bytes, and produces, by means of complicated and boring math, an output of some shorter number of bytes.
    # For example, the string `hello` ends up with a hash of `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`.
    # If we passed in, say, a 2GB movie file, the hash function would look at all the bytes, and compute a hash that is the same length as the one above for `hello`. In fact, SHA256 is named because its output is 256 bits in length.
- Since only this relatively short hash is stored, we can compare files much more quickly.
- This is faster because each file is read only once as it get hashed, and only the short hashes need to be compared to see if the files are the same.
- There is still a chance of collisions, or different inputs yielding the same outputs, since we are ultimately reducing an infinite number of files to a finite number of hashes. But 2^{256} is such a large number of files that the chance of this happening is essentially zero.
    # David later emails a friend at Dropbox to inquire, and it turns out that they store files in 4MB chunks, with each chunk hashed, and if there happens to be some chunks with the same hashes, then only one of them is kept.
- The other implication is that Dropbox gets to look through all your files, even if it's encrypted while it's being sent or stored.
- And a good hash function should have the property of having a completely different hash between two files that are only slightly different, so that changes are more obvious.
- In real life, people on the internet (including us[5]) sometime post hashes of files that other people download, so people can verify that their download has the same hash.

[5] http://cdn.cs50.net/2014/fall/lectures/12/m/

- Also, given an outputted hash, it is theoretically impossible to find the input file. To do this, someone would generally have to try every possible input, until they found one that produced the given hash.
- Finally, one more feature of Dropbox is streaming sync. This means that when we upload a new file, someone else can download it as soon as we start, rather than we finish. This essentially means that the sync takes about half as long. Normally a file would be sent to some server as a whole, and only when that's done can someone else start downloading it. But Dropbox uploads files in chunks, so someone can download those chunks as they are uploaded.

Code

- Last time we watched a TED video about algorithms[6].
- Some of its pseudocode looked like this:

```
let N = 0
for each person in room
  set N = N + 1
```

- `N` is called a **variable**, where we use a letter or word to denote a container in which we can store values. In this case, `N` is like a counter that stores a number.
- In the third line, notice that we're first evaluating the right side, `N + 1`, and then putting that in what's on the left side, `N`, to get the desired effect of increasing `N` by one.
- The second line is called a **loop**, which is like a cycle, or something that happens again and again.
- A **condition**, or branch, will handle certain cases. In the second example in the video, when counting pairs of people in a room, the case of having an odd number of people is handled by such a condition:

```
if 1 person remains then
  set N = N + 1
```

The line that is indented will only occur if the condition is true.

[6] https://youtu.be/6hfOvs8pY1k

- Now we can transition to real code, or **source code**, which is what programmers write.
- A language called C has source code that might look like this:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- There's a lot of syntax here, but we're going to start with a language called Scratch, which is a lot more accessible.
- The end result of the source code is a lot of ones and zeroes, and that's what computers actually run in their processor.
- To get this end result, or **object code**, we use a program called a **compiler** that converts source code, C, to instructions that a CPU understands, like add or move or set.
- Here is another program, written in Python, that does the same thing:

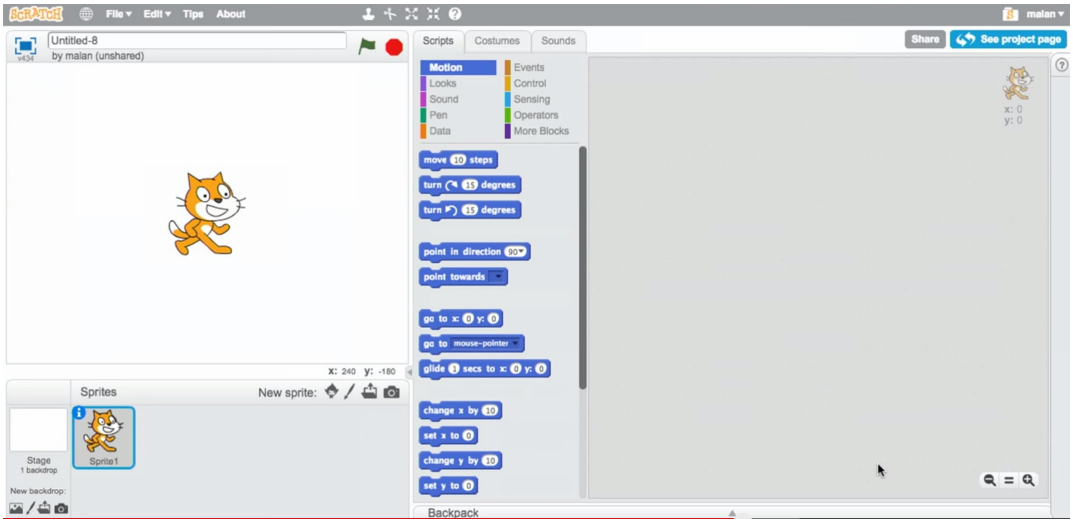
```
print "hello, world"
```

- Python is an **interpreted language**, as opposed to C, a **compiled language**. This means that source code for Python isn't actually runnable, but rather we run another program called an **interpreter** that reads in the source code and runs it on the CPU line by line. Because of this, interpreted languages tend to be a little slower, since they need to be converted as they are running, rather than ahead of time.

Scratch

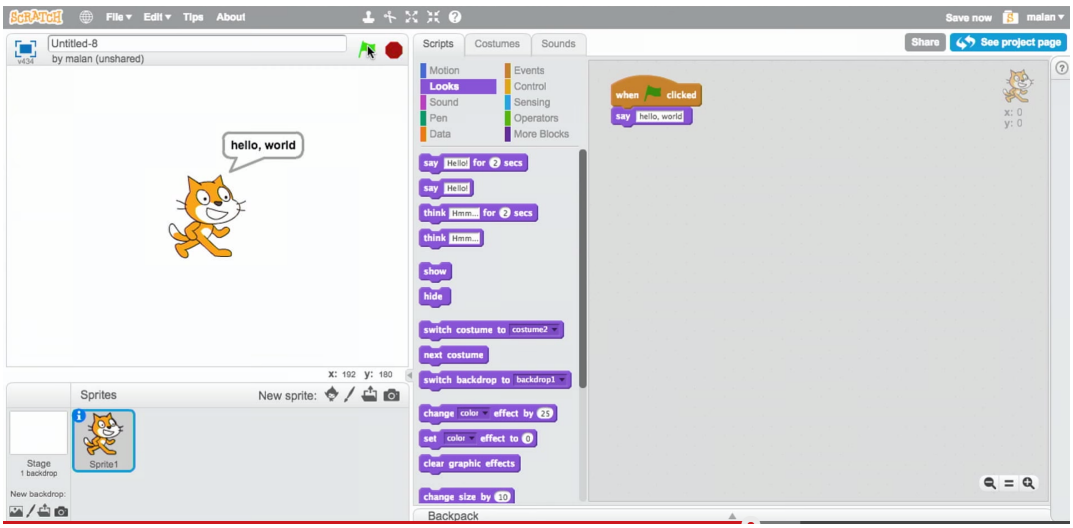
- We'll start with an interpreted language that's entirely graphical: Scratch[7].
- We go through a demo of the editor, which looks like this:

[7] http://scratch.mit.edu



The blue blocks (and others, selectable with the colors above them) can be dragged and dropped to the area on the right make the cat do things.

- When we drag the `[ when (green flag) clicked ]` block from the Events category to the area on the right right, then a `[say [] ]` block from the Looks category underneath it, and then click the green flag, we get something that looks like this:

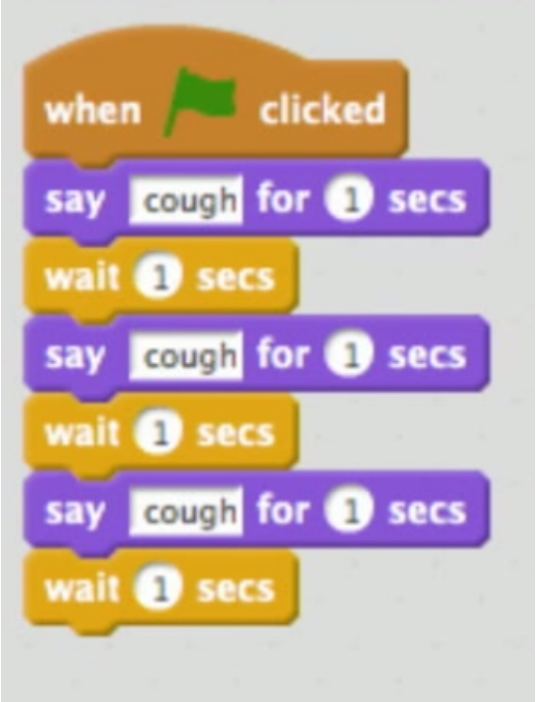


- Here is another sample project[8] created by dragging and dropping more blocks together.
- We can call each block that does something a **statement**. For example, the `[say [] ]` block is a statement. `[wait]` is another block that's a statement.

[8] https://scratch.mit.edu/projects/26329254/

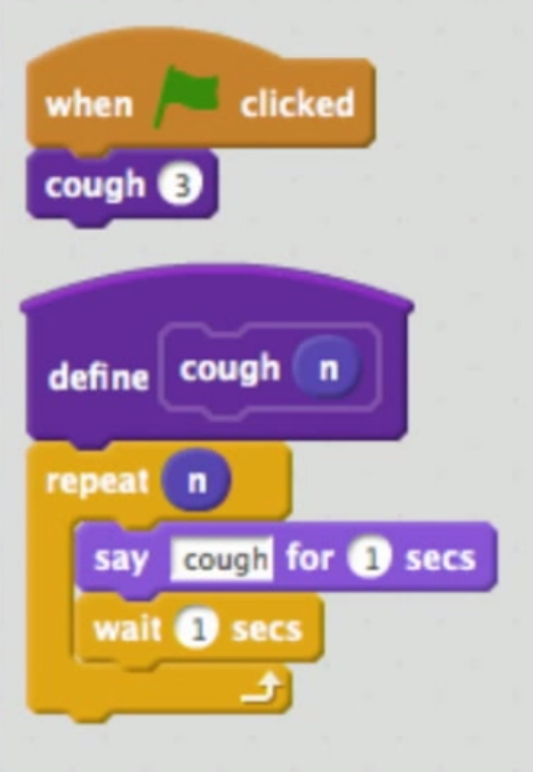
- A **boolean expression** is either `true` or `false`. `< mouse down? >`, `< [x] < [y] >`, `< touching [mouse-pointer] ? >` are all boolean expressions.
- We can combine them with blocks like `< < x > and < y > >`, where the entire block is only true if both `< x >` and `< y >` are individually true.
- These boolean expressions are particularly useful in **conditions** such as the `[ if < > ]` block. There, only if the expression in the hexagon shape is true, will the statements in the bottom part of the `if` block be executed.
- A similar block is the `[ if < > ] else` block, where the condition leads to a fork in the road, where one set of statements will be executed if it's true, and another if it's not.
- We can also nest them, with an `[ if < > ] else` block within the `else` of another, and this would allow for three forks in our road.
- Loops can be created with a `[ forever ]` block among others, and even though sometimes we want programs to end on their own, other times we might want something to continue forever until we tell it to stop.
- A block like `[ repeat [x] ]` will repeat some set of statements `x` number of times.
- We can also interact with variables with blocks like `[ set [N] to [0] ]`.
- We can define functions with the `[ define ]` block, which just means that a set of blocked can be grouped together and named, and then used as a single block, so we don't have to click and drag the same blocks over and over again to implement some functionality.
- We demonstrate some more programs[9], each with different functions and effects.
- In particular, the `cough` program has blocks that look like this:

[9] https://scratch.mit.edu/studios/1070446/



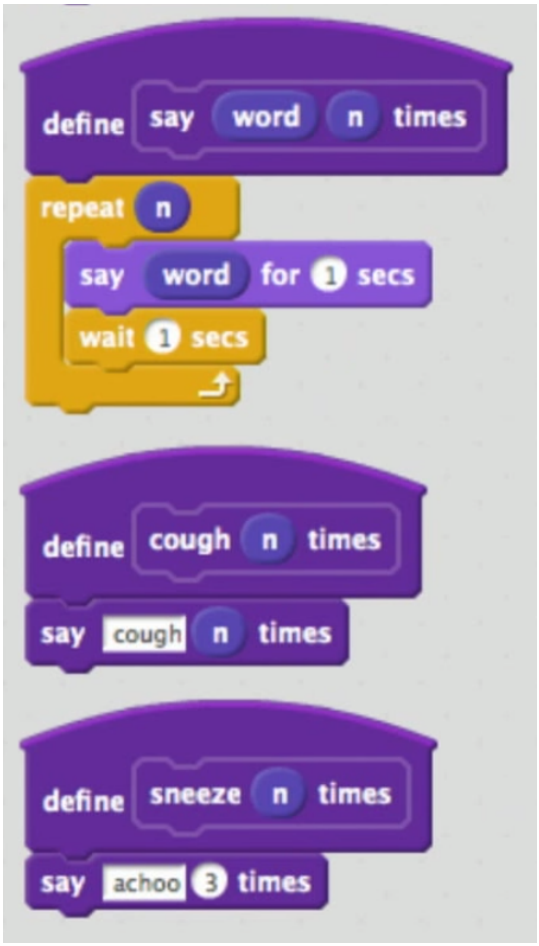
- Notice that some pattern of block is repeated three times, and though this is correct in that it works as expected, the design could be improved as follows:

- Now the code is cleaner, more readable, and can be changed more easily, and though this might seem to be a huge difference here, it certainly can be in larger programs.
- While Scratch takes care of keeping track of how many times we've been through a loop, so we only `cough` 3 times as we intend, in languages like C we have to create a variable ourselves to keep count.
- The function `cough` is also an example of **abstraction**, which means we've essentially hidden how the function works behind a function name that we can just call from our main function, without worrying about how it's implemented.
- We can also do something fancy and **parameterize** our `cough` function:




```
# This just means we can call it with some additional parameter that makes it do something slightly different. In this case, we can cough any number of times.
```

- We can go even further, and abstract functions like `cough` and `sneeze`, that essentially just `say` a word `n` times, like so:



The `say [word] (n) times` block now takes two inputs, a `word` and a number `n`, and does that, so we can easily create more functions that make noise.

- In Scratch, the idea of an array is called a list, and we can use blocks to add items to these lists.
- Scratch also supports the idea of **threads**, or multiple things happening at once. (To put it simply, the CPU just switches between threads really quickly, to make it appear as though everything is happening simultaneously, even though it might only be able to do one thing at a time.)
- **Events** can also occur, where one sprite (an object in Scratch, like the cat) can talk to another.

David goofed! He meant to show the example called `events`. `hi hi hi` actually used shared state, or variable sharing across sprites.