Week 6

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Introduction

- Today's lectures is being broadcast from Yale!
- David reassures everyone that you can handle CS50.
    - A 3 on a problem set is not a 60%! It really is a good grade — we meant that when we said it at the beginning of the semester.
    - What matters most is where you end up in Week 12 relative to yourself in Week 0.
    - Our less comfortable vs. more comfortable tracks aren't separated for the quiz grading statistics, so don't worry too much about where you are relative to the mean/median.
    - We normalize grades across sections at the end of the semester, we take into account less vs. more comfortable tracks, and on top of that, we take an incredibly personalized approach to grading each individual student (so try to be a little bit patient at the end of the semester if grades are on the late side - this process takes time!).

  - And we’ll repeat it one more time: most of you should be getting 3s on the problem sets! Particularly at the beginning of the semester, we try to make sure there’s room left at the top of the scale, so we can see how you improve over the course of the semester. A 3 is a good grade; a 4 is great; a 5 is excellent.

2. Transitioning to Web Programming

- Let’s take a look at what happens under the hood of the Internet with Warriors of the Net[1], an animated short film that goes into detail about how information is packed into packets and sent across the Internet.
- Today we transition from lower-level C programming to higher-level web programming.
- As an aside: Apparently there’s a men’s bathroom at Yale called the "Harvard Room". We’ll have to get on creating an analog in Cambridge…

3. Data Structures Recap

- Looking back at where we left off last week, with some data structures that could be of use for Problem Set 5:
  - We could use an array or a linked list to store our dictionary, but an array might be tricky to build since we don’t know in advance how many words are in our dictionary, and a linked list limits us to linear-time search.
  - A hash table (with separate chaining) gives us O(log *n*) search time, assuming that we use a good **hash function** that spreads elements out well over the table. (Technically the asymptotic time doesn’t care what hash function you use, because the difference is just a constant, but it can be a very large constant — imagine a million items all linked together in one linked list vs. spread out over a thousand buckets — and it will certainly make a difference to your real-time running time in seconds.)
  - A trie gives us constant time lookups (well, lookups proportional to the length of the word, but the length of words in your dictionary should always be less than the number of words in your dictionary)! It takes up a lot of memory, though, most of which just contains a bunch of `NULL` pointers.

[1] http://youtu.be/PBWhzz_Gn10

- The concept of **hashing** that we brought up in the context of hash tables has many more applications.
  # Hashing simply means using a hash function to convert some kind of input to a number.
  # One hash function that's been used for a while in the real world is **SHA1**, which converts whatever data you have (any sequence of 0s and 1s, which can be MB or even GB in length) into just a few bits, so that you have a number you can work with.
  # This hash function has been used to secure bank transactions, cell phone encryption, and all kinds of other security applications.
  # We're getting close to the "SHAppening"[2] - researchers have recently published that it's possible crack SHA1 (reverse-engineer its input from its output) by applying only about $175,000 worth of compute power.
  # Fortunately, we knew this was going to happen, and most of the world has moved on to using SHA256 — a version that produces longer hashes — for security purposes. CS50's own website uses SHA256.

4. Announcements

- Problem Set 5 is out.
- Quiz 0 is this week.
- OHs[3] available in preparation for the quiz.
- Problem Set 5 has a reputation for being the hardest problem set in this course, but that means that we're almost at the peak of the course, and you should get a tremendous feeling of satisfaction knowing that you've completed this.
  # Students in most intro CS courses haven't implemented trees, tries, or hash tables by the time they finish.
- There are only four more problem sets, so all is soon to be green fields and puppies :-)

2 http://sites.google.com/site/itstheshappening
3 http://cs50.harvard.edu/hours

5. Web Programming!

- As we transition to web programming, we're getting into a realm that we're all more familiar with actually interacting with — in a web browser rather than a text prompt.
- Within your home, you might have a laptop or two, maybe an old-school desktop, and they all access the Internet via a network hub, which talks to a modem, which connects you to the outside Internet.
- So what happens when you actually connect to the Internet? Every computer connected to the Internet needs a unique address, just like houses that want to receive mail need postal addresses.
- Where do these addresses come from? There's a special server on most networks called a **DHCP server**, which stands for Dynamic Host Configuration Protocol, which assigns an address each time a new computer joins the local network.
- These unique addresses to identify each device on the Internet are called **IP addresses**, and they take the form of `#.#.#.#`, where each number is in the range of 0-255.
  # Each number, then, uses 8 bits, and in total the address is 32 bits, making for a total number of roughly 4 billion possible addresses.
- Though 4 billion is a high number, we have lots of servers and devices, which is starting to be problematic.
  # We're starting to run out of addresses, and although this problem has also been foreseen for a long time (and there are solutions out there), we haven't collectively switched over to better systems particularly rapidly.
    # This is similar to many other situations in the tech world, where problems are known for a long time before they're solved (e.g., the Y2K bug had been recognized for many, many years, and yet we were rushing to fix it at the last minute).
  # The system that we're finally, slowly converting to is called **IPv6** (the current system is IPv4), and instead of 32 bit addresses (4 billion possibilities), IPv6 uses 128 bit addresses (300 billion billion billion billion possibilities).
  # Specific ranges are also reserved for particular organizations or providers. For example, many of the computers at Yale have IP addresses in the ranges

130.132.#.# or 128.36.#.#, while computers at Harvard are often assigned an IP address that starts with 140.247.#.# or 128.103.#.#. Providers like Comcast also has a particular prefix that they own.

- Addresses that start with 10.#.#.#, 172.16.#.# - 172.31.#.#, or 192.168.#.# are **private IP addresses** that we have set aside to use within a particular network, but not on the Internet at large.
  - Private IP addresses have been one way we've dealt with the problem of running out of addresses: we can give a whole bunch of computers a single IP address to the rest of the world by putting a device in front of them (called a **router** or **proxy**) that talks to the rest of the Internet using that one IP address, while within the local network, computers each have their own private addresses (which can be repeated between different local networks).
  - This means that, for example, websites visited by two computers behind the same router can't tell the difference between the two visitors - all of your data is going through the router with the same IP address.
- In our quest to ruin more media for you, let's watch a quick (inaccurate) clip[4] from the TV show Numb3rs[5] that shows us "how the Internet works."
- The last frame of that clip is a screen of code with a reference to `crayon`, suggesting that the code that the "hacker" is using is probably just some drawing program.
- Additionally, the top of the screen shows `http://275.3.6.28` in the address bar, which is still wrong since it's not a valid IP address (probably to keep viewers from visiting a real server).
- But how often do we type numeric addresses into our browsers? It's a pain to remember IP addresses, so we use words, kind of like how companies will often have a phone number that spells out something memorable rather than expecting their customers to remember their phone number on its own.
- The system that converts IP addresses to words and vice versa is called the Domain Name System, or **DNS**. A DNS server basically stores a list of domain names and IP addresses that correspond to them.
- After some Internet difficulties, let's open a Terminal window and do the following:

[4] http://youtu.be/5ceaqtWhdnI
[5] https://en.wikipedia.org/wiki/Numbers_(TV_series)

```
% nslookup yale.edu
Server:   130.132.1.9 ❶
Address:  130.132.1.9#53 ❷

Name: yale.edu
Address: 130.132.35.53 ❸
```

# The first part, lines 2-3, is the address of Yale’s DNS servers, and the last line, line 6, is its response to the question of what the Yale website’s IP address really is. And when we copy that number, and go to `http://130.132.35.53`, we indeed end up at yale.edu.

# But this is much less memorable than "yale.edu", which is why we use human-friendly domain names rather than IP addresses directly.

- If we did the same with google.com, we get this:

```
% nslookup google.com
Server:   130.132.1.9
Address:  130.132.1.9#53

Non-authoritative answer:
Name: google.com
Address: 74.125.226.68
Name: google.com
Address: 74.125.226.67
Name: google.com
Address: 74.125.226.78
Name: google.com
Address: 74.125.226.65
Name: google.com
Address: 74.125.226.71
Name: google.com
Address: 74.125.226.72
Name: google.com
Address: 74.125.226.70
Name: google.com
Address: 74.125.226.73
Name: google.com
Address: 74.125.226.69
Name: google.com
```

```
Address: 74.125.226.64
Name: google.com
Address: 74.125.226.66
```

  - Sometimes companies tell the world they have one IP address, which ends up being resolved, or mapped, to a whole bunch of servers after, or, like Google, they tell the world that there are a number of addresses, any of which you can contact.
  - But Yale doesn't have just one server, nor does Google have only ten or so; they use a technique called "load balancing" to spread out requests across their many (in Google's case, probably thousands) servers.

- So that's what's been happening under the hood when you type in the name of a website: your operating system asks the DNS server what the address of this website is.
- We still need to get our actual data to and from the Internet. For this, we use a **router**, which is in charge of "routing" stuff: sending packets, or envelopes of digital information, from sender to receiver.
- There are many thousands, probably millions of routers around the world. If we want to send a message from a computer in New Haven to one in Cambridge, we don't have a cable that goes directly there — instead, we have a router here that's connected to many other routers, which uses a table in its memory to figure out which other router to send the message to next.

5.1. Traceroute

- We can actually see the routers that our messages go through.
- If we wanted to see the servers between us and MIT, we can type `traceroute -q 1 www.mit.edu`, which runs the `traceroute` program in quiet mode, once, to MIT's website:

```
% traceroute -q 1 www.mit.edu
traceroute to www.mit.edu (23.209.73.59), 64 hops max, 52 byte packets
 1  arubacentral-vlan30-router.net.yale.internal (172.28.204.129) 36.353
 ms
 2  10.1.2.13 (10.1.2.13)  1.464 ms
 3  cen10g-asr.net.yale.internal (10.1.4.30)  2.953 ms
 4  jfk2-edge-02.inet.qwest.net (65.124.208.93)  35.206 ms
 5  nyc2-edge-02.inet.qwest.net (205.171.134.46)  4.183 ms
 6  host-216-206-47-190.dia.qwest.net (216.206.47.190)  4.041 ms
 7  a23-209-73-59.deploy.static.akamaitechnologies.com (23.209.73.59)
 4.701 ms
```

- Each of these rows is like a student in the audience between David and Dan, who passed the message along.
- On line 2 we see the domain name typed in, and `23.209.73.59` is apparently the IP address of `www.mit.edu` that the computer figured out using DNS. We're also going to limit ourselves to `64 hops`, or by going through no more than 64 servers between us and MIT.
- Then each row is a router, with the first being the amusingly named Yale central router "arubacentral", which appears to be on a virtual local area network (VLAN).
- The next router has no name, just some private (because it starts with 10.#.#.#) IP address.
- In the middle, steps 4 & 5, we go to routers with "jfk" and "nyc" in their names. It's common to name routers after the nearest airport code or major cities.
- Finally, the actual domain name for `www.mit.edu`, seems to indicate a server that's part of a company called Akamai that they've outsourced server hosting to. This means that their website might not even be hosted in Cambridge (although as it turns out, Akamai is also based in Cambridge).

- Let's go further to `www.cnn.co.jp`, CNN in Japan:

```
$ traceroute -q 1 www.cnn.co.jp
traceroute to www.cnn.co.jp (14.0.33.140), 64 hops max, 52 byte packets
 1  arubacentral-vlan30-router.net.yale.internal (172.28.204.129) 36.353
 ms
 2  10.1.2.13 (10.1.2.13)  1.471 ms
 3  cen10g-asr.net.yale.internal (10.1.4.30)  2.932 ms
 4  te-4-1.car1.stamford1.level3.net (4.26.48.81)  3.619 ms
 5  *
 6  124.215.192.77 (124.215.192.77)  79.800 ms
 7  otejbb205.int-gw.kddi.ne.jp (203.181.100.137)  180.785 ms
 8  sjkBBAC07.bb.kddi.ne.jp (106.162.175.154)  188.651 ms
 9  obpBBAC03.bb.kddi.ne.jp (111.87.242.70)  192.322 ms
10  111.86.159.66 (111.86.159.66)  185.208 ms
11  14.0.40.86 (14.0.40.86)  187.124 ms
12  14.0.42.95 (14.0.42.95)  184.472 ms
```

  # In step 5, we can see that one of the routers didn't send us a full response (usually for privacy reasons).

  # If we look at steps 7 and 8, it suddenly took a lot longer to get a response from those servers - because we've crossed an ocean!

  # But here, despite the number of hours it would take to fly to Japan, our message took under than 200 milliseconds to send.

- So you can play around, and some servers might give you a `*` as an answer for privacy's sake, but generally you can see the route your message takes. It doesn't always take the shortest path, as the path it takes is a function of whatever dynamic routing decisions the system makes at the time you send the message.
- There's a huge amount of undersea cable to carry Internet data around the world, as shown in this clip[6].

5.2. TCP/IP

- Let's go to a slightly lower level and think about how our data is actually represented as it's transferred around the world.
- When you request a webpage, or send or receive an email, or any other kind of data on the Internet, it doesn't come in one huge chunk of bits - instead, it's broken up into a bunch of **packets**.
- To demonstrate this, we have a picture of Rick Astley that we want to send to volunteer Cole in the back of the lecture hall. In human terms, we'd say something like, "Can you pass this to Cole?" and pass it along until it finally reached him.

  # This is a really big piece of data, and we don't want to stop all the other traffic on the Internet, so we're going to break it up into smaller pieces to send to Cole. We place each one into different envelopes, labeling them 1 of 4, 2 of 4, 3 of 4, and 4 of 4. Each envelope is addressed to Cole, and says it's from David.

- Now we can hand out each of the envelopes, even to separate routers, and in theory all four should make their way to the back of the lecture hall (even though they won't necessarily take the same route on the way there).
- Then Cole can reassemble the picture and realize that there should be 4 pieces.

[6] http://youtu.be/KWxwYbaAWxs#t=35m32s

  - But what if one router is broken or is powered off, and a packet doesn't make it to Cole?
  - **TCP**, or **Transmission Control Protocol**, is another protocol used with IP on the Internet (you may have seen **TCP/IP**), which guarantees delivery. TCP tells computers to send a packet back - a message from Cole to David - telling him which packets in the original message were missing, since they were all numbered.
- But in reality, just addressing those packets to "Cole" wouldn't be sufficient. First, we'd use Cole and David's IP addresses, rather than their names. And second, there are lots of services on the Internet (email, chat, file transfers, etc), not just webpages, and we need to know which data goes to which service.
- With TCP, we have a set of conventional **port numbers** associated with certain services:
  - 21 FTP
    25 SMTP
    53 DNS
    80 HTTP
    443 HTTPS
  - For example, **FTP**, file transfer protocol, was assigned a unique identifier of 21 some years ago.
  - **SMTP**, for outbound email, is 25.
  - And you may have seen that HTTP, web traffic, and HTTPS, secure web traffic, use 80 and 443. (Our web browsers append these port numbers to the ends of domain names for us so we don't have to specify which port, but if you append `:80` to most web addresses, you'll end up in the same place.)
    - The number for HTTPS can be greater than 255 because they have to do with TCP, not IP (which is 4 numbers, 0-255). A port number in TCP is a separate 16-bit integer value, so in theory can be really big, but in practice under a few thousand.
- So how does a **firewall** work? A firewall is a device that all Internet traffic at a particular location (say, Yale, or a company) has to go through, which blocks access to certain sites by blocking outgoing requests to certain IP addresses or ranges.
  - We could imagine trying to prevent access to a site by changing its listing in our DNS server to an incorrect IP address, but then someone could gain access by going

directly to the correct IP address, or by changing their DNS server settings (e.g., to 8.8.8.8, Google’s DNS server).

  # Note that if you use Google’s DNS servers, Google can see every site you access (since you’re literally asking them what the IP address is for every URL you visit).

• You may have heard about the "Great Firewall of China", and students traveling abroad can get around it by using a **VPN**, or **virtual private network** (such as access.yale.edu for Yale students or vpn.harvard.edu for Harvard students), which gives you an encrypted connection between your computer and a server elsewhere.

  # This lets you circumvent firewall restrictions, because all your traffic first goes to this server elsewhere, and where it goes after that isn’t visible to the router that’s maintaining the firewall because it’s encrypted.

  # You can also use a VPN if you’re using unencrypted WiFi, to prevent someone else on the unsecure network from listening to all your traffic.

  # This does slow your connection down, since there could be a significant distance between you and the VPN server, and between the VPN server and the site you’re trying to access.

5.3. HTTP Requests

• What do we actually use all this network infrastructure for?

• You’re probably familiar with the abbreviation **HTTP**, just from seeing it at the beginning of URLs; it stands for **Hypertext Transfer Protocol**. This is the language, or protocol, that web browsers and web servers use to talk to each other.

• A protocol is just a set of conventions, often for communication. For example, when you introduce yourself to another human, you likely shake hands with them and say your name; this is a protocol in our society, and computers have various protocols of their own.

• Consider the following picture (it’s a bit dated as you can tell by the appearance of the computers):

  # The client is your machine that asks for information, and the server is the machine that responds with information.

- **GET** is the type of request that a browser sends to a server to, well, get information. It makes a request in the form of a textual message that literally says something like this:

```
GET / HTTP/1.1
Host: www.google.com
...
```

  - This simple message would be opened by the server on the other side, which then responds accordingly.
  - The `/` right after `GET` is just asking for the root directory, or the highest directory. To properly visit a website, we should really be typing `http://www.facebook.com/` with that final `/` meaning we want the root of the hard drive, or the default page.
  - The next part, `HTTP/1.1`, means that we're using version 1.1 of HTTP to talk to the server.
- So now we get something like this back as a response:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

  - The first line is confirming that we're using version 1.1 of HTTP to communicate, and `200` is a status code that means `OK`: the server has the page we're looking for.
    - There are lots of other status codes that you're more likely to have seen (since they represent errors that your browser will tell you about, rather than just serving the page), 404 Not Found being maybe the most obvious, but there are many others:
    - 200 OK
      301 Moved Permanently
      302 Found
      401 Unauthorized
      403 Forbidden
      404 Not Found
      500 Internal Server Error

  # The second line is telling the web browser that you’re getting back a webpage of the type `text/html` (as opposed to an image or video, for example).
  # And then the `…` is the actual message that the server responds with.

- Our requests can contain other information as well, though: if you turn off "instant results" on Google and search for "cats", the resulting URL might look like this:

```
https://www.google.com/search?
site=&source=hp&q=cats&oq=cats&gs_l=hp.3..0i131i46j46i131l2j0l2j0i131j0j0i131j0j0i3.146
```

- Wow, that looks really crazy. But we can distill it down to just:

```
https://www.google.com/search?q=cats
```

  # …and we still get the exact same page of results.
  # This is equivalent to sending the following request:

```
GET /search?q=cats HTTP/1.1
Host: www.google.com
...
```

- This is how web servers, the computers running the software that makes websites work, take input from users. More generally, after the URL, we can pass a list of key, value pairs like this:

```
https://www.website.com/page?
key1=value1&key2=value2&anotherkey=anothervalue
```

- But GET is not the only way to send requests. GET has a major disadvantage: every parameter has to be added to the end of the URL.
  # What if we’re logging into a website and we type in our password? We don’t want that to show up in the address bar (where it’s visible to someone walking behind us, it’ll be remembered by our browser, and it’s visible to the DNS server).
  # If we’re uploading a photo, or some other kind of information that doesn’t lend itself well to representation as a text string, we might have a problem.
- So instead, we can use a **POST** request, which looks much the same:

```
POST /login.php HTTP/1.1
Host: www.facebook.com
```

```
...
email=malan@harvard.edu&password=12345
```

# This gives the inputs `email=malan@harvard.edu`[7] and `password=12345` to Facebook's `login.php` (the program, written in PHP, that allows users to log in). But now, instead of those parameters being part of the URL as they were before, they're "inside the envelope", so to speak — they're in the message rather than in the header.

# Assuming Facebook is using HTTPS, this is encrypted, making it a secure way to pass things like passwords to the web server.

5.4. HTML

- What's actually inside all these packets we've been sending and requesting using TCP and HTTP? Data that the web browser can understand, to turn into web pages the user can view, is encoded in **HTML**, or **Hypertext Markup Language**.
- Let's look at this bit of HTML:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

# It does nothing other than display `hello, world`, and we notice that the first line declares this piece of code as using HTML, followed by various tags beginning with `<` and ending with `>`.

- Let's go to a text editor, and save a file titled `hello.html` somewhere simple, like the Desktop.
- Then we can start with something like this:

[7] mailto:malan@harvard.edu

```
<!DOCTYPE html>

<html>

</html>
```

# Notice that we type out `</html>`, closing the tag, ahead of time, and remember to put everything else inside those tags.

- Next we make a section called `<head>` that every HTML page has, and add a `<title>`:

```
<!DOCTYPE html>

<html>
  <head>
    <title>hello, world</title>
  </head>
</html>
```

- Every page also has a `<body>` section, which we can add here, like this:

```
<!DOCTYPE html>

<html>
  <head>
    <title>hello, world</title>
  </head>
  <body>
    hello, world
  </body>
</html>
```

- We can save it, and even though the file will not be on a server but just our own computer, we can double-click on the file to open it in Chrome:

# Notice that the body is the large white space, and the title is at the top of the tab.

- HTML is not a programming language like C (it won't have any loops, conditions, etc), but a **markup language**, that tells the browser how to display information.

- We’ll use **PHP**, which is a programming language, to dynamically generate HTML.
- We can represent this HTML as a tree:

```
<!DOCTYPE html>                                  (document)
                                                     |
<html>                                             [html]
  <head>                                           /    \
    <title>hello, world</title>                [head]  [body]
  </head>                                        |        |
  <body>                                      [title]  (hello, world)
    hello, world                                 |
  </body>                                    (hello, world)
</html>
```

- But this website we’ve just created is pretty mind-numbingly boring. Just the text "hello, world" in black on a white background. We can make it more exciting using **CSS** — also not a programming language, but a system of syntax by which we can change the appearance of our webpages.
- And again, continuing to ruin TV and movies for you, one more clip[8] about how hacking works on the Internet (because clearly you can type better if you have two people at the keyboard at once).

[8] http://youtu.be/KWxwYbaAWxs#t=68m07s