Day 1

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

Cheng Gong

Table of Contents

Gangnam Style

- The view count for Gangnam Style on YouTube[1] did exceed the capacity of a 32-bit number. This morning, it had 2,280,759,099 views, but 2,147,483,647 is the biggest decimal number that fits into a 32-bit integer (where all but the sign bits are 1s). The counter didn't actually roll over back to 0, though, because YouTube had anticipated this problem, fixed it, and created an animation that made the counter look like it was breaking, as an easter egg[2].
- 2^32 is actually around 4 billion, but a 32-bit signed integer can go up to a maximum of 2 billion, since half of the values it can represent are negative numbers.
- David texted some friends at YouTube because he was so ashamed of himself for falling for this story, and they told him about how it was fixed ahead of time.

[1] https://www.youtube.com/watch?v=9bZkp7q19f0
[2] http://en.wikipedia.org/wiki/Easter_egg_%28media%29

Internet Issues

- Last night, some people had trouble accessing the website, and we managed to chase down the bug.
- **HTTP Strict Transport Security (HSTS)** is a new technology for web browsers that allow websites to tell the world that they support HTTPS, or encrypted, secure connections, and that browsers should only connect using encryption.
- This was all fine and dandy, but if you type in `http://cs50.harvard.edu/mba` (or even just `cs50.harvard.edu/mba`, as most browsers these days automatically fill in the `http://` part), the server should respond with a message like `HTTP/1.1 200 OK`. We'll look at this message more closely next week, but our server returned a message of:

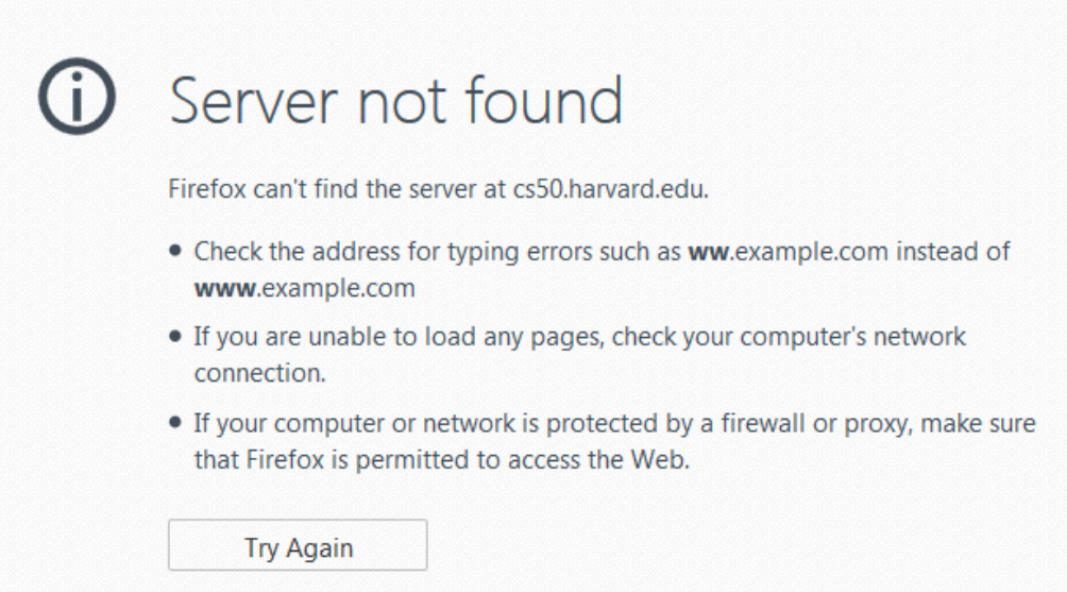
```
http://cs50.harvard.edu/mba

HTTP/1.1 301 Moved Permanently
Location: https://cs50.harvard.edu/mba
```

- which means what it sounds like: our website's moved to another URL, which is the same apart from the `s` added to `http`.
  # We did this on purpose to automatically redirect people to the encrypted version of the website.
  # But the web server automatically redirected us again, since `/mba` didn't point to a page, but rather a directory, and needed a `/` at the end:

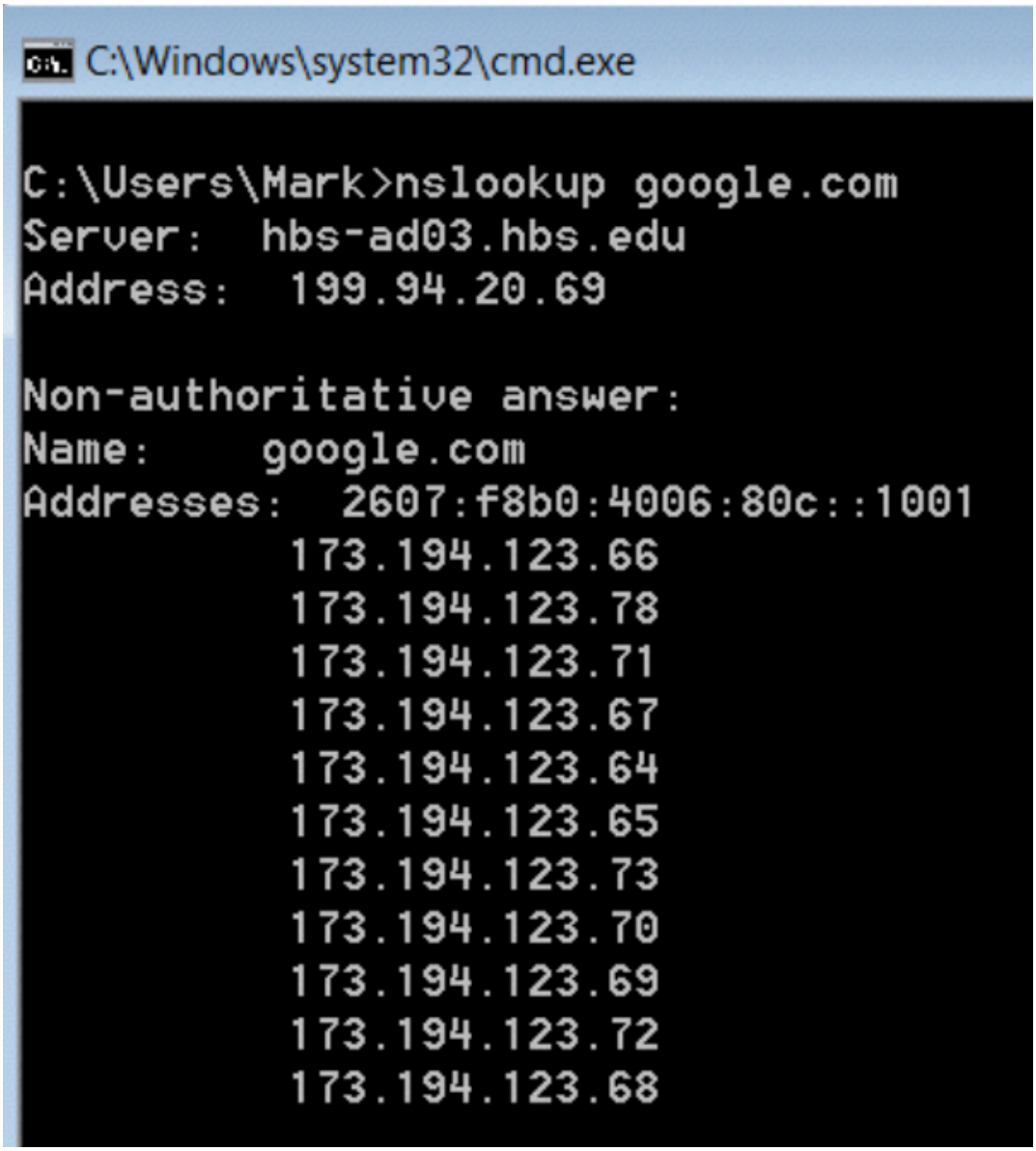
```
https://cs50.harvard.edu/mba

HTTP/1.1 301 Moved Permanently
Location: http://cs50.harvard.edu/mba/
```

- And now we've unintentionally redirected you to an unencrypted version, but since we already used HSTS to enforce only using `https`, browsers couldn't connect to our website.
- Long story short, we tinkered with the server settings until it stopped telling browsers to go back and forth, from secure to insecure and insecure to secure.

- To make things more complicated, other people had errors that looked like this:

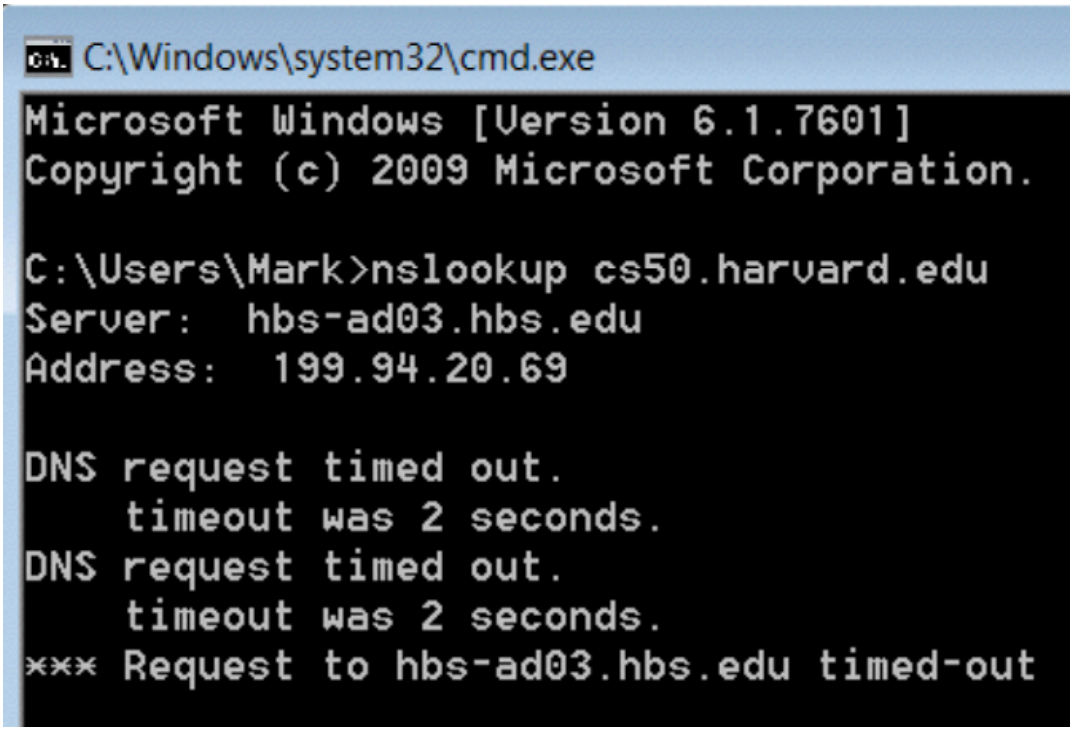


- We'll talk about this more next week, too, but it turns out that URLs like `google.com` are all mapped to numbers called IP addresses, and there are servers in the world that convert URLs to numbers.
- We can run a program like `nslookup`, name server lookup, to see these responses:



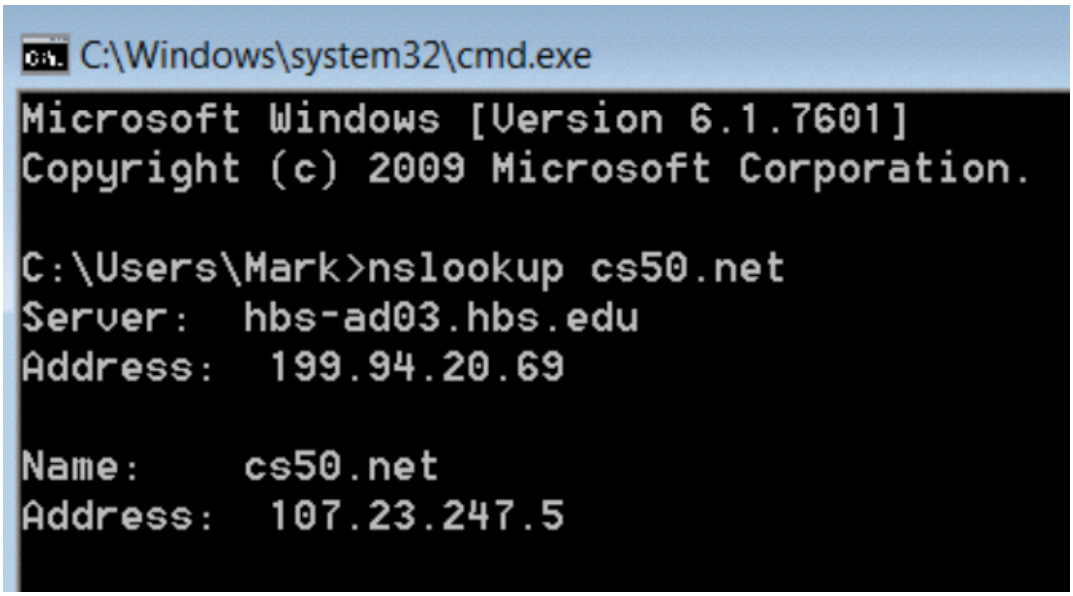
It looks like a server named `hbs-ad03.hbs.edu` responded to us, and it told us that the IP address for `google.com` is any of those. Google has lots of servers doing the same thing, so talking to any of them would work.

- But when we tried to lookup our own website, we got this:



Timing out just meant that our request took too long, and we didn't get an answer back. Our browsers were basically making the same request and getting the same result back, which is why we got the `Server not found` error earlier.

- For some reason, going to `cs50.net` worked:



    # Someone from IT services helped us by figuring out that HBS's internet servers basically had outdated information compared to FAS's servers from across the river, so once that was fixed, the problem should be fixed.
    # (`cs50.net` was the official URL before we got the `cs50.harvard.edu` domain name, so it still works to keep old links alive.)

Course info

- We'll have office hours soon, to help with assignments or projects or just chat.
- In Assignment 0 we asked a few questions, and we've compiled the statistics, so you can be assured you're not alone.
- In particular, 45% of students in the class would describe themselves as less comfortable, 43% in between, and 12% more comfortable.
- Only 1% of students say they have a lot of programming experience, with 45% having none and 53% having a little.
- 62% of people also have taken no prior CS courses, with 18% having taken 1, 16% 2, and 4% 3.
- Interestingly, many people have had some exposure to HTML, with SQL, MATLAB, Java, and JavaScript following behind in popularity.
- People are also split on what they want their team sizes to be, so there are opportunities to find teammates.
- In terms of potential seminars, we've received a good list:
    # A/B testing
    # artificial intelligence
    # biometrics
    # **case studies**
    # communicating with engineers
    # data analysis
    # excel macros
    # **mobile app development**

    # **modern technologies and platforms**

    # **security**

    # statistical software

    # UI design

    # virtual reality

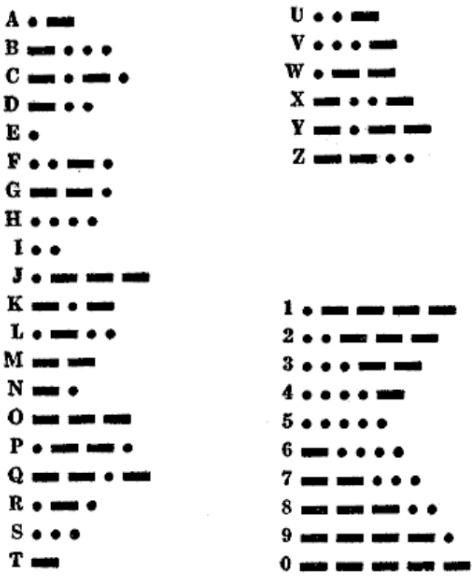
    # **virtualization**

    # **web development**

- We'll cover the bolded topics in class, and try to schedule some seminars on the other topics.
- We'll also start providing a set of notes and video recordings for each of the days on the classes[3] page.

Day 0, recap

- Looking at a chart of:



    # what might a sequence of 6 dots represent?

[3] http://cs50.net/mba/classes/

    # A classmate proposes that it could be `HI`, given that there could be a pause after the first four dots, but if there were no pauses, it could be `SS` or `5E`, and lead to some ambiguity.
    # Remember that we avoid this problem in ASCII by having every character have a fixed length of 8 bits!
- The binary represenation for the decimal number 42 is `00101010`, and though the two zeroes at the beginning are not necessary, we add them to get 8 bits, by convention.
- Hexadecimal is base-16, so 42 is `2A`. 2 times the 16s place is 32, and A is 10, since to represent 16 digits we use 0-9 and then A-F, making for that total of 42.
- There are other popular base systems, with Base64[4] being one of them. Emails, for example, are just text, so attachments like photos and videos are converted to those typable symbols first, and sent that way.
- `hello, world` in ASCII is `104 101 108 108 111 044 032 119 111 114 108 100`.

Peanut butter jelly time

- David's brought along some bread, peanut butter, and jelly, so we can implement a proper algorithm together.
- Someone suggests the first step is to get the plate.
- Next, we open the bag of bread.
- Step 3, put a piece of bread on the plate.
- Step 4, "open the peanut butter," was not specific enough. Telling David to "remove the lid" was also insufficient, but "unscrew the peanut butter lid" worked. We also needed to remove the foil seal from the peanut butter, as a part of this step, though it could have been a separate step.
- Step 5, pick up the knife. David grabs the knife by the blade.
- Step 6, put the knife in the peanut butter. David puts the knife in the peanut butter, handle first.
- Step 7, twirl the knife.
- Step 8, scoop some peanut butter out with the knife. David pulls a chunk out with the knife.

[4] http://en.wikipedia.org/wiki/Base64

- We never finish making the sandwich, but we see how lack of precision or corner cases can really change the meaning of what we say.
- 42 is the "Answer to the Ultimate Question of Life, The Universe, and Everything" from The Hitchhiker's Guide to the Galaxy[5], a popular series, with lots of other creative answers from classmates.
- People also had various questions of their own:
  # What is the minimum level of CS background required for this class? Really, none. The material might not be easy or familiar, but many classmates also have no experience, so we can get through it together.
  # Can we have case studies? The class will not focus primarily on business cases, though we'll certainly include technical examples when we can.

Sorting

- Yesterday we solved the problem of finding a name in the phone book under the condition that it was already sorted.
- If we have a number of doors, behind which are numbers randomly placed, we'd have to look one by one behind all the doors to find a particular number.
- But if the numbers were sorted, we could use the method of **binary search** from last time, where we look in the middle, compare the value there to what we want, and look either in the left half or the right half, and so on.
- Now the question is how we might sort our data. A classmate shuffles a standard deck of cards, and then a classmate at the front attempts to sort them.
- Our volunteer is putting the cards in order as he gets them, and moving them around as more cards are found.
- The general approach a human would take is to probably sort by number or suit first, like making four piles of suits first, and then within each suit by putting each number where it might go on the table as you get them, or finding the lowest and highest card first, and repeating.
- Let's take a look at the following 8 numbers:

```
4    2    6    8    1    3    7    5
```

[5] http://en.wikipedia.org/wiki/The_Hitchhiker%27s_Guide_to_the_Galaxy

    # We'll also start calling such lists **arrays**, just things put next to each other like in a list.

- We'll have 8 volunteers hold up each of these numbers at the front of the room.
- A classmate suggests looking for the smallest number first, if we wanted to sort these numbers.
    # A computer would need to start at the first number, look at each number in the list one at a time, and remember where the smallest number is, after going through the entire list.
- Then we put the smallest number, `1`, at the front of the list, swapping it with `4`, so the list looks like:

```
1    2    6    8    4    3    7    5
```

    # Since the other numbers are in random order already, in general we'd probably not make the problem any worse by putting `4` where `1` was.

- Moving all the numbers down one slot and putting `1` at the beginning is also correct, but takes more time than just swapping the two. Moving `4` to the end would take up more memory, as now we need more space, unless we moved the other numbers up, in which case we're again using more time.
- Then we start at the second slot in the list, and note that `2` is the smallest in the rest of the list, and leave it there.
- So we can continue this algorithm:

```
4    2    6    8    1    3    7    5
1    2    6    8    4    3    7    5    // 1 is moved toward the front
1    2    3    8    4    6    7    5    // 2 is moved toward the front
1    2    3    4    8    6    7    5    // 3 is moved toward the front
1    2    3    4    5    6    7    8    // 4 is moved toward the front
```

- The list was actually sorted after we swapped `5` and `8`, but for complete correctness we'd have to have checked the list from `6` forward, then from `7` forward, and then from `8` forward.
- We can take another approach, where we start with the shuffled list again:

```
4    2    6    8    1    3    7    5
```

- But now we just start at the first two elements, swapping them if they're out of place:

```
2   4   6   8   1   3   7   5
```

- And we continue by going down and comparing numbers one pair at a time, swapping them as necessary:

```
2   4   6   1   8   3   7   5
2   4   6   1   3   8   7   5
2   4   6   1   3   7   8   5
2   4   6   1   3   7   5   8
```

- But even though we've reached the end of the list, the most we know is that the last number is in the right place, since even if it started at the far left it would have moved all the way to the right, as it's the largest number. Everyone else would only have moved one place at most.
- So now we start over at the beginning, and do the same thing again:

```
2   4   6   1   3   7   5   8
2   4   1   6   3   7   5   8
2   4   1   3   6   7   5   8
2   4   1   3   6   5   7   8
```

- Now both 7 and 8 are correct. One more time:

```
2   1   4   3   6   5   7   8
2   1   3   4   6   5   7   8
2   1   3   4   5   6   7   8
```

- And again:

```
2   1   3   4   5   6   7   8
1   2   3   4   5   6   7   8
```

- After we swapped `1` and `2` this time, the list was sorted, but we could only know that by going down the entire list and checking each pair, remembering if we had made any changes. If we had, then we need to start over again to be completely sure, but if we hadn't, then the list must be sorted and we're done.
- The first algorithm that we used, where we selected the smallest element over and over again, is **selection sort**. (A variation is to select the largest element each time.)
- The second algorithm is called **bubble sort**, where the largest elements bubbled up to the right, and the smallest elements bubbled to the left.

- Another way to sort these numbers is called **insertion sort**. First, we start with the first number, and it's automatically sorted since we only have the one number so far, and we also remember that everything after `4` is unsorted. (Imagine a line between `4` and `2`.)

```
4    2    6    8    1    3    7    5    // 4 is sorted
```

- Now we take the next number, `2`, and put it in the right place in our sorted list, which is now `2` and `4`, with our imaginary line moved to between `4` and `6`.

```
2    4    6    8    1    3    7    5    // we move 2 to in front of 4
```

- We repeat this until the entire list is sorted:

```
2    4    6    8    1    3    7    5    // 6 is sorted
2    4    6    8    1    3    7    5    // 8 is sorted
1    2    4    6    8    3    7    5    // we move 1 to the beginning
1    2    3    4    6    8    7    5    // we move 3 to its location
1    2    3    4    6    7    8    5    // we move 7 to its location
1    2    3    4    5    6    7    8    // we move 5 to its location
```

- Instead of going down the list over and over again, we only need to go down it once, but every time we need to place a number in the correct place in our sorted part of the list, we needed to move the other numbers down, one at a time. And that, too, depends on how out-of-order the numbers are, so to speak. In the case of moving `1`, we had to move several numbers down, but in the case of moving `7`, we only needed to move `8` down.

Running Time

- So let's think about the benefits and costs of each of these algorithms, to compare them.
- Let's use a variable, `n`, to represent the size of a given array of numbers.
- If we think about our first algorithm, selection sort, we needed `(n - 1)` comparisons to find where the smallest number was.
- After we put that in the right place, we only needed `(n - 2)` comparisons to find the smallest number, since we already know the first number is sorted. We repeat, until we get to the last two numbers, where it only took one step to find the smallest number, for a total of

```
(n - 1) + (n - 2) + ... + 1
```

```
# steps.
```

- Doing some algebra, this simplifies to:

```
(n - 1) + (n - 2) + ... + 1
(n^2 - n)/2
n^2/2 - n/2
O(n^2)
```

- So now we can say that this algorithm takes on the **order** of `n^2` steps, since as `n` increases in size `n/2` becomes insignificant compared to the term with `n^2`.

```
# If we had a million numbers, we see this is true:
```

```
n^2/2 - n/2
1,000,000^2/2 - 1,000,000/2
500,000,000,000 - 500,000
499,999,500,000
```

- This is formally called **Big O notation**[6], which is what computer scientists would use to describe running time and efficiency of algorithms.
- In addition, the exact number of steps doesn't need to be accounted for, as we are looking for asymptotic running time, or how the running time of the algorithm grows as the size of *n* increases. For example, the constant part of `n^2/2` can be thrown away, leaving us with just $O(n^2)$, since the part of the expression that has the biggest effect when *n* is really large is `n^2`.
- Bubble sort is also n^2, since it takes us roughly *n* steps to go over the entire list, and we do it roughly *n* times, for a total of roughly n^2 steps.
- Examples of algorithms that take roughly *n* steps (meaning they are #(*n*)) are linear search or counting people in a room one at a time. Counting people two at a time is still fundamentally the same speed, making it #(*n*), because as *n* gets really large the number of steps still increase the same way.
- The green line from yesterday, with the curve, is different. An algorithm with #(log *n*) steps is binary search, since it divided the problem in half over and over.
- An algorithm that takes constant time, regardless of the size of the problem, has #(1) time. For example, doing something with just the first element of a list, or a statement that just does something like "leave the room."

[6] http://en.wikipedia.org/wiki/Big_O_notation

- The opposite is **big Omega**, Ω, to refer to a lower bound of how many steps an algorithm might take.
- An algorithm that takes $\Omega(n)$ steps is finding the smallest element in an array, since we need to look at all *n* elements to be sure. Taking attendance, too, requires a minimum of *n* steps.
- Something that takes $\Omega(1)$ steps is anything that takes a constant number of steps to just do something, like saying hi or leaving the room.
- As an aside, if # and Ω happen to be same for some algorithm, then we say it is in **Theta**, or Θ, of some value, as in $\Theta(n)$ for counting the number of elements in an array.
- To see how our sorting algorithms work, you can play with this demo[7]. (Except you need Java, so you really should just watch the lecture video and save some effort!)
- If we set the demo to run an algorithm called merge sort against selection sort and bubble sort, we see that it's much faster.

  # In fact, even a certain senator can comment[8] on the inefficiencies of bubble sort.
- We can sort in *O*(*n* log *n*) with merge sort. Remember that, with the phone book, to fundamentally solve the problem we used a divide-and-conquer approach.
- Merge sort does this, where it sorts *n* elements, but only log *n* times instead of *n* times.
- Let's look at the pseudocode:

```
On input of n elements
    if n < 2
        return
    else
        sort left half of elements
        sort right half of elements
        merge sorted halves
```

- First, we check the size of the list, since a list with one or zero elements is already sorted. That's our **base case** (previously with the phonebook it was either calling Mike or giving up) which ensures that the algorithm will eventually stop.
- The next part with the `else` is a loop, but notice that we say we'll sort the halves, without really specifying how. Except, if we sort each half with the same algorithm, eventually we'll get to a half that's just one element, and the looping will stop.

[7] http://cg.scs.carleton.ca/~morin/misc/sortalg/
[8] http://www.youtube.com/watch?v=k4RRi_ntQc8

- For merging, given two lists, we'll start at the beginning of both, and take the smaller element of the two lists to start our sorted list, and continue until all the elements are used. We'll only need to look at each person once, too, since each half is already sorted.

  # We'll need twice as much memory, since we'll need space to put the elements into our new sorted list, but we'll save the time of having to move elements down.

- Let's see an example of this:

```
4    2    6    8    1    3    7    5
```

- There are 8 elements, so we mentally divide the list in two halves.

```
4    2    6    8    1    3    7    5
```

- Then we start over, and again proceed to the `else` block, which means we are sorting the left half of the left half:

```
4    2    6    8    1    3    7    5
```

- We call the algorithm again, but this time we have just one element, 4, which is sorted. We move on to 2, which is also sorted by itself:

```
4    2    6    8    1    3    7    5

4    2    6    8    1    3    7    5
```

- Now the left half is sorted and the right half is sorted, so now we can merge the two halves, taking whichever element comes first and placing it on the left:

```
4    2    6    8    1    3    7    5    // list
2    4                                  // now the left half of the left
 half is sorted
```

- So we continue our sort of the left half, where now we have to sort the right half of the left half.

```
4    2    6    8    1    3    7    5    // list
2    4    6    8                       // now the both halves of the left
 half is sorted
```

- And now we merge the left half, but it happens to already be sorted so we don't see any change:

```
4    2    6    8    1    3    7    5    // list
2    4    6    8                        // now the both halves of the left
 half is sorted
```

- We return to the right half and quickly repeat the same process:

```
2    4    6    8    1    3    7    5

// sort the left half of the right half
2    4    6    8    1    3    7    5

// sort the left half of the left half of the right half
2    4    6    8    1    3    7    5

// sort the right half of the left half of the right half
2    4    6    8    1    3    7    5

// merge the left half of the right half
2    4    6    8    1    3    7    5

// sort the left half of the right half of the right half
2    4    6    8    1    3    7    5

// sort the right half of the right half of the right half
2    4    6    8    1    3    7    5

// merge the right half of the right half
2    4    6    8    1    3    5    7

// merge the right half
2    4    6    8    1    3    5    7
```

- Finally, we need to merge the original left half with the original right half:

```
4    2    6    8    1    3    7    5
2    4    6    8    1    3    5    7
1    2    3    4    5    6    7    8
```

- One of the themes in computer science is having tradeoffs. Sometimes, we can spend more memory space to achieve a faster running time, as we do here.
- That was a lot to keep in mind, but let’s try to think about how long merge sort takes in term of running time.

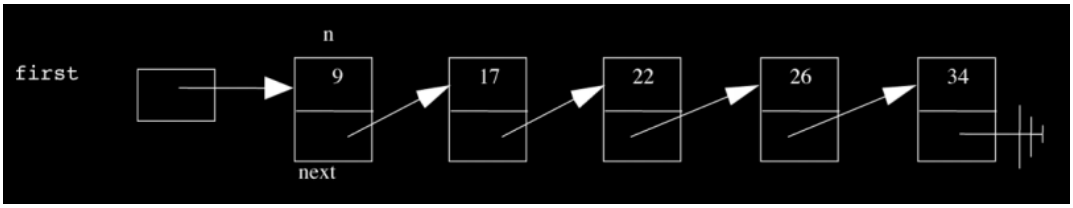
- Remember that we divided in half every time, so there were three divisions, or $\log_2 n$, and at every division we looked at all n elements, for a total of n log n steps.
- Another approach is to look back at the pseudocode:

```
On input of n elements
    if n < 2
        return
    else
        sort left half of elements
        sort right half of elements
        merge sorted halves
```

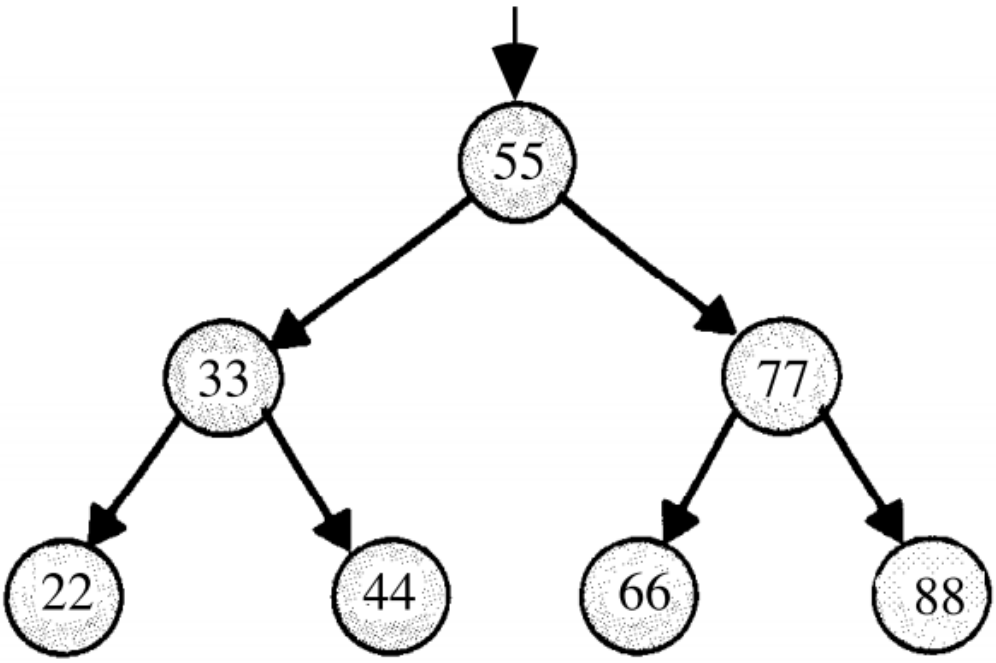
  # The first condition is constant time, so it's on $O(1)$ time. We can say $T(n) = O(1)$ if $n < 2$, as in the time it takes to solve a problem of size n is constant if n is less than two.
  # The second part simplifies to this: $T(n) = T(n/2) + T(n/2) + O(n)$ if $n >= 2$, since we needed to sort both halves, and merge them.
  # Although it's not obvious, that function actually simplifies to $O(n \log n)$.
- There are lots of other sorting algorithms, the fastest ones of which run in $O(n \log n)$ time, at the cost of some space.

Data Structures

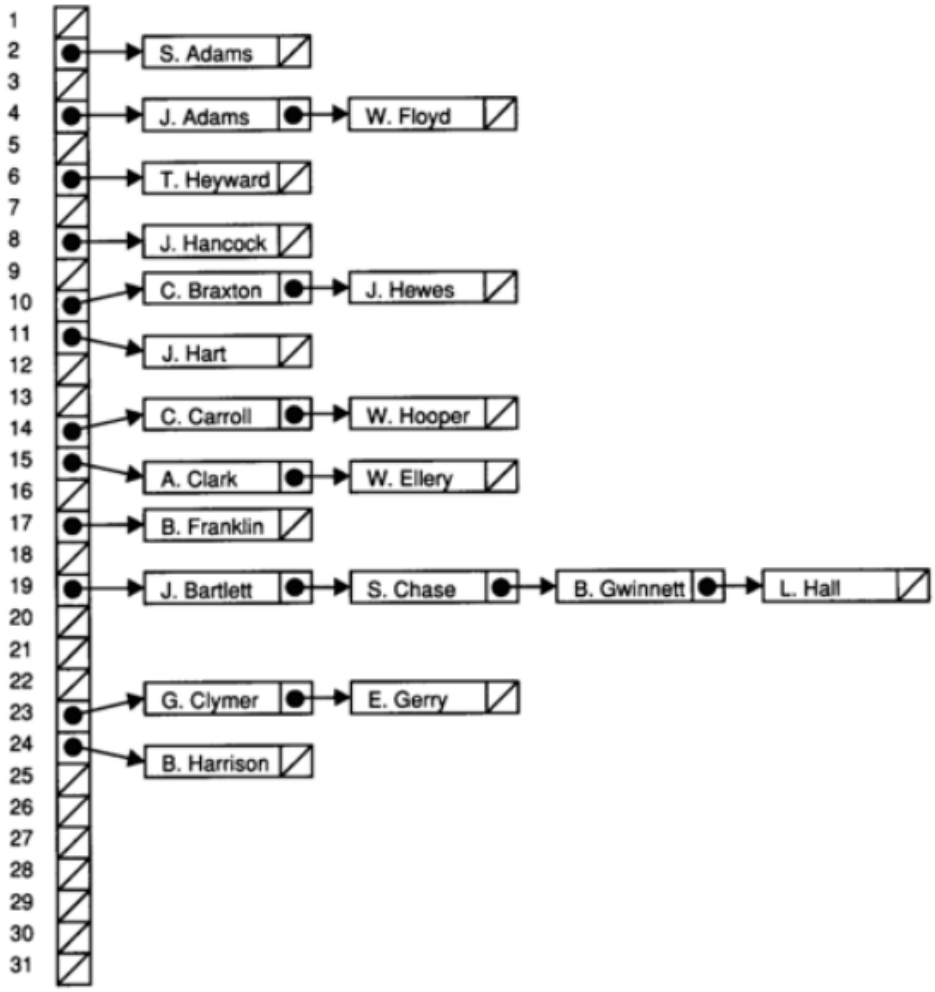
- An **array** has some number of objects, but they're all next to each other, and are the same type of object. This means we can jump to any index in the array and have **random access**. **RAM** in your computer is called **random access memory** for this reason: since each chunk of memory is the same size and next to each other, we can jump to any chunk we want with a little math. For example, if our array had 32-bit integers, the first one would start at bit 0, the second at bit 32, then 64, and so on.
- But if we allocate a certain number of elements at the beginning, we can't add more later, since there might be other things stored in memory after the end of the array. Alternatively, you could try to ask for space to fit both the array and a new element, but you'd have to copy everything over to that new space, with $O(n)$ time.
- We can avoid this problem somewhat by allocating more memory than we need at the very beginning, but this uses extra memory that we might never need.
- We can solve this problem altogether by a different structure called a **linked list** that looks like this:



- The numbers here don’t have to be contiguous, so we can have gaps between them, with arrows pointing from one chunk of memory to the next.
- The arrows are called **pointers**, and we don’t need to go too much into detail in this class, but we should notice that in a linked list, adding an extra number means that all we need to do is to get one more chunk of memory, and add an arrow at the end pointing to that new chunk, without any copying involved.
- But now we don’t have random access, since we need to follow each arrow from the beginning to get to any element in the list.
- Along with the theme of tradeoff, you almost never get anything for free in computer science, or engineering more generally.
- Yet another data structure is a **binary search tree**:



- Instead of having each chunk of memory have an arrow to the next linearly, we could also conceptually lay them out in the shape of a tree, with two arrows.
- The numbers to the left of each item are always less, and the numbers to the right are always greater.
- If there are *n* nodes in this tree (**node** being a general item in memory), it only takes us *O*(log *n*) time to find any item, since the tree is sorted and we divide by two for every level we go down.
- Another data structure is called a **stack**, where the most recently placed item is taken out first. This property is also called **LIFO**, last-in-first-out. You can think of this as a literal stack of trays, where clean ones are placed on top of a pile, and people take the ones on the top as they come in.
- A line outside an Apple Store would be the opposite, since the first person should be taken first (**FIFO**, first-in-first-out). The corresponding data structure is appropriately a **queue**.
- The fanciest we'll talk about is a **hash table**:



- Suppose we have a whole bunch of names, and we want to store them somewhere where we can find them easily. We can combine two ideas, an array and linked list. Let’s say that each person has a day of the month that they were born on, from 1 to 31, so we can make an array of size 31, so we can look people up by their birthdays quickly. But we don’t know how many people are born on each day in advance, so we make each day point to a linked list, so now if we need to find a person we won’t need to search a list containing all the names we know, but just the list of people born on that day.

  # More concretely, in our example, to find the name of someone, we have to first **hash** them by finding their birthday somehow. In general, hashing is just converting an input into an index, usually numerical, to look them up in our hash table’s array.

- The big *O* running time of this is *O*(*n*) still, because we could theoretically have a list of people whose birthdays fall on the same day, and need to find them in the linked list with linear search. But most likely we'll have a roughly even distribution, which will lead us to have an average of *O*(*n*/31) steps.
- In the real world, we also might not have billions or trillions of inputs, so theoretical and practical running times could be a lot different.
- A larger hash table, with more than 31 chains in the array, would have a running time that's faster in practice.
- And an ideal hash table is one whose chains are all length one, meaning we get to our data in just one step. But we need an ideal hash function for this, as in a function that converts each name to a unique index. Right now, our hash function is just the day of birth, which is far from ideal since many people can share the same day. Finding a hash function that spreads our data out into many chains of roughly equal length is the hard part!
  - # Using a social security number is a great idea since that's unique, but since there are 9 digits, that means our hash table needs to have 10^9 entries, and that's a lot of space.
- We end on this demo[9] of what various sorting algorithms sound like.

[9] https://www.youtube.com/watch?v=t8g-iYGHpEA