This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Lecture 3

Searching

- Last time, we talked about memory in a computer, or RAM, and how our data can be stored as individual variables or as arrays of many items, or elements.
- We can think of an array with a number of items as a row of lockers, where a computer can only open one locker to look at an item, one at a time.
- For example, if we want to check whether a number is in an array, with an algorithm that took in an array as input and produce a boolean as a result, we might:
  - look in each locker, or at each element, one at a time, from the beginning to the end.
    - This is called **linear search**, where we move in a line, since our array isn't sorted.
  - start in the middle and move left or right depending on what we're looking for, if our array of items is sorted.
    - This is called **binary search**, since we can divide our problem in two with each step, like what David did with the phone book in week 0.
- We might write pseudocode for linear search with:

```
For i from 0 to n-1
    If i'th element is 50
        Return true
Return false
```

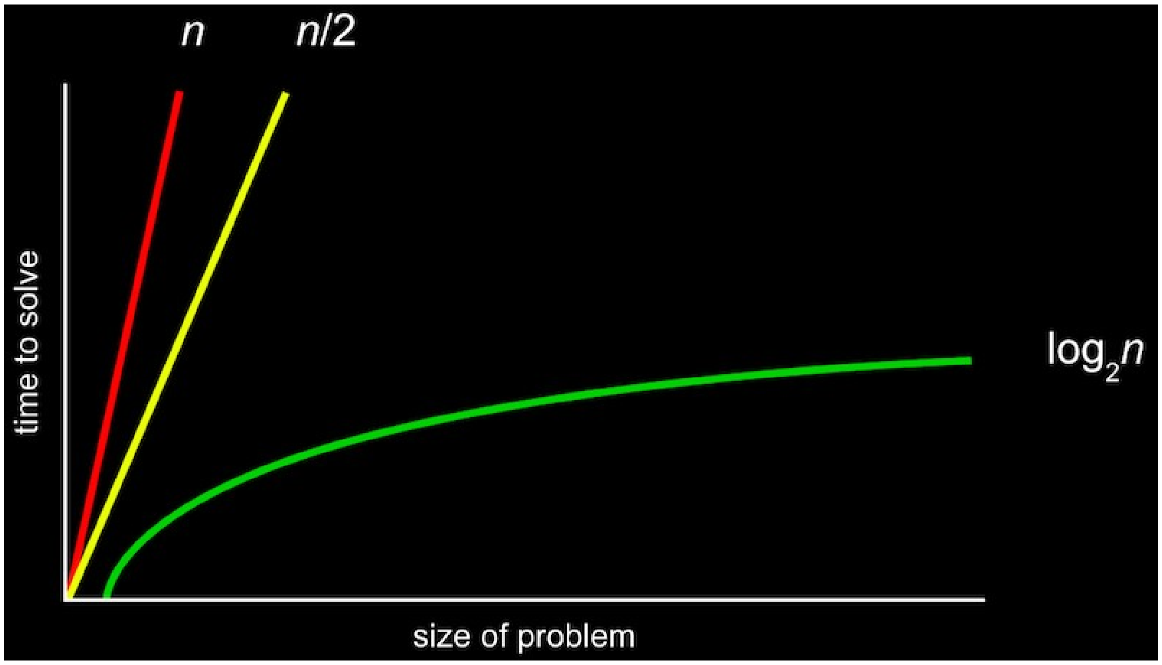
  - We can label each of `n` lockers from `0` to `n-1`, and check each of them in order.
- For binary search, our algorithm might look like:

```
If no items
    Return false
If middle item is 50
    Return true
Else if 50 < middle item
    Search left half
Else if 50 > middle item
    Search right half
```

- Eventually, we won't have any parts of the array left (if the item we want wasn't in it), so we can return `false` .
- Otherwise, we can search each half depending on the value of the middle item.

Big O

- In week 0, we saw different types of algorithms and their running times:



- The more formal way to describe this is with big *O* notation, which we can think of as "on the order of". For example, if our algorithm is linear search, it will take approximately *O*(*n*) steps, "on the order of *n*". In fact, even an algorithm that looks at two items at a time and takes *n*/2 steps has *O*(*n*). This is because, as *n* gets bigger and bigger, only the largest term, *n*, matters.
- Similarly, a logarithmic running time is *O*(log *n*), no matter what the base is, since this is just an approximation of what happens with *n* is very large.
- There are some common running times:
  - $O(n^2)$
  - *O*(*n* log *n*)
  - *O*(*n*)
    - (linear search)
  - *O*(log *n*)
    - (binary search)
  - *O*(1)
- Computer scientists might also use big Ω, big Omega notation, which is the lower bound of number of steps for our algorithm. (Big *O* is the upper bound of number of steps, or the worst case, and typically what we care about more.) With linear search, for example, the worst case is *n* steps, but the best case is 1 step since our item might happen to be the first item we check. The best case for binary search, too, is 1 since our item might be in the middle of the array.
- And we have a similar set of the most common big Ω running times:
  - $\Omega(n^2)$

- Ω(*n* log *n*)
- Ω(*n*)
  - (counting the number of items)
- Ω(log *n*)
- Ω(1)
  - (linear search, binary search)

Linear search

- Let's take a look at `numbers.c` :

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    // An array of numbers
    int numbers[] = {4, 8, 15, 16, 23, 42};

    // Search for 50
    for (int i = 0; i < 6; i++)
    {
        if (numbers[i] == 50)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - Here we initialize an array with some values, and we check the items in the array one at a time, in order.
  - And in each case, depending on whether the value was found or not, we can return an exit code of either 0 (for success) or 1 (for failure).
- We can do the same for names:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // An array of names
    string names[] = {"EMMA", "RODRIGO", "BRIAN", "DAVID"};

    // Search for EMMA
    for (int i = 0; i < 4; i++)
    {
        if (strcmp(names[i], "EMMA") == 0)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - We can't compare strings directly, since they're not a simple data type but rather an array of many characters, and we need to compare them differently. Luckily, the `string` library has a `strcmp` function which compares strings for us and returns `0` if they're the same, so we can use that.

- Let's try to implement a phone book with the same ideas:

```c
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string names[] = {"EMMA", "RODRIGO", "BRIAN", "DAVID"};
    string numbers[] = {"617-555-0100", "617-555-0101", "617-555-0102", "617-555-0103"};

    for (int i = 0; i < 4; i++)
    {
        if (strcmp(names[i], "EMMA") == 0)
        {
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - We'll use strings for phone numbers, since they might include formatting or be too long for a number.
  - Now, if the name at a certain index in the `names` array matches who we're looking for, we'll return the phone number in the `numbers` array, at the same index. But that means we need to particularly careful to make sure that each number corresponds to the name at each index, especially if we add or remove names and numbers.

Structs

- It turns out that we can make our own custom data types called **structs**:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

typedef struct
{
    string name;
    string number;
}
person;

int main(void)
{
    person people[4];

    people[0].name = "EMMA";
    people[0].number = "617-555-0100";

    people[1].name = "RODRIGO";
    people[1].number = "617-555-0101";

    people[2].name = "BRIAN";
    people[2].number = "617-555-0102";

    people[3].name = "DAVID";
    people[3].number = "617-555-0103";

    // Search for EMMA
    for (int i = 0; i < 4; i++)
    {
        if (strcmp(people[i].name, "EMMA") == 0)
        {
            printf("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

- We can think of structs as containers, inside of which are multiple other data types.
- Here, we create our own type with a struct called `person` , which will have a `string` called `name` and a `string` called `number` . Then, we can create an array of these struct types and initialize the values inside each of them, using a new syntax, `.` , to access the properties of each `person` .
- In our loop, we can now be more certain that the `number` corresponds to the `name` since they are from the same `person` element.

Sorting

- If our input is an unsorted list of numbers, there are many algorithms we could use to produce an output of a sorted list.
- With eight volunteers on the stage with the following numbers, we might consider swapping pairs of numbers next to each other as a first step.
- Our volunteers start in the following random order:

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

- We look at the first two numbers, and swap them so they are in order:

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

- The next pair, `6` and `8`, are in order, so we don't need to swap them.
- The next pair, `8` and `5`, need to be swapped:

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

- We continue until we reach the end of the list:

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

- Our list isn't sorted yet, but we're slightly closer to the solution because the biggest value, `8`, has been shifted all the way to the right.
- We repeat this with another pass through the list:

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

  - Note that we didn't need to swap the 3 and 6, or the 6 and 7.
- Now, the next biggest value, `7`, moved all the way to the right. If we repeat this, more and more of the list becomes sorted, and pretty quickly we have a fully sorted list.
- This algorithm is called **bubble sort**, where large values "bubble" to the right. The pseudocode for this might look like:

```
Repeat n-1 times
    For i from 0 to n-2
        If i'th and i+1'th elements out of order
            Swap them
```

  - Since we are comparing the `i'th` and `i+1'th` element, we only need to go up to $n - 2$ for `i`. Then, we swap the two elements if they're out of order.
  - And we can stop after we've made $n - 1$ passes, since we know the largest n−1 elements will have bubbled to the right.
- We have $n - 2$ steps for the inner loop, and $n - 1$ loops, so we get $n^2 - 3n + 2$ steps total. But the largest factor, or dominant term, is n^2, as `n` gets larger and larger, so we can say that bubble sort is $O(n^2)$.
- We've seen running times like the following, and so even though binary search is much faster than linear search, it might not be worth the one–time cost of sorting the list first, unless we do lots of searches over time:
  - $O(n^2)$
    - bubble sort
  - $O(n \log n)$
  - $O(n)$
    - linear search
  - $O(\log n)$
    - binary search

  - *O*(1)
- And Ω for bubble sort is still n^2, since we still check each pair of elements for $n - 1$ passes.

Selection sort

- We can take another approach with the same set of numbers:

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

- First, we'll look at each number, and remember the smallest one we've seen. Then, we can swap it with the first number in our list, since we know it's the smallest:

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

- Now we know at least the first element of our list is in the right place, so we can look for the smallest element among the rest, and swap it with the next unsorted element (now the second element):

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

- We can repeat this over and over, until we have a sorted list.
- This algorithm is called **selection sort**, and we might write pseudocode like this:

```
For i from 0 to n–1
    Find smallest item between i'th item and last item
    Swap smallest item with i'th item
```

- With big *O* notation, we still have running time of $O(n^2)$, since we were looking at roughly all *n* elements to find the smallest, and making *n* passes to sort all the elements.
- More formally, we can use some formulas to show that the biggest factor is indeed n^2:

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

- So it turns out that selection sort is fundamentally about the same as bubble sort in running time:
  - $O(n^2)$
    - bubble sort, selection sort
  - $O(n \log n)$
  - $O(n)$
    - linear search
  - $O(\log n)$
    - binary search
  - *O*(1)
- The best case, Ω, is also n^2.
- We can go back to bubble sort and change its algorithm to be something like this, which will allow us to stop early if all the elements are sorted:

```
Repeat until no swaps
    For i from 0 to n–2
        If i'th and i+1'th elements out of order
            Swap them
```

- Now, we only need to look at each element once, so the best case is now Ω(*n*):
    - $\Omega(n^2)$
        - selection sort
    - Ω(*n* log *n*)
    - Ω(*n*)
        - bubble sort
    - Ω(log *n*)
    - Ω(1)
        - linear search, binary search
- We look at a visualization online comparing sorting algorithms (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) with animations for how the elements move within arrays for both bubble sort and selection sort.

Recursion

- Recall that in week 0, we had pseudocode for finding a name in a phone book, where we had lines telling us to "go back" and repeat some steps:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If Smith is on page
5      Call Mike
6  Else if Smith is earlier in book
7      Open to middle of left half of book
8      **Go back to line 3**
9  Else if Smith is later in book
10     Open to middle of right half of book
11     **Go back to line 3**
12 Else
13     Quit
```

- We could instead just repeat our entire algorithm on the half of the book we have left:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If Smith is on page
5      Call Mike
6  Else if Smith is earlier in book
7      **Search left half of book**
8
9  Else if Smith is later in book
10     **Search right half of book**
11
12 Else
13     Quit
```

    - This seems like a cyclical process that will never end, but we're actually dividing the problem in half each time, and stopping once there's no more book left.
- **Recursion** occurs when a function or algorithm refers to itself, as in the new pseudocode above.
- In week 1, too, we implemented a "pyramid" of blocks in the following shape:

```
#
##
###
####
```

- And we might have had iterative code like this:

  ```
  #include <cs50.h>
  #include <stdio.h>

  void draw(int h);

  int main(void)
  {
      // Get height of pyramid
      int height = get_int("Height: ");

      // Draw pyramid
      draw(height);
  }

  void draw(int h)
  {
      // Draw pyramid of height h
      for (int i = 1; i <= h; i++)
      {
          for (int j = 1; j <= i; j++)
          {
              printf("#");
          }
          printf("\n");
      }
  }
  ```

  - Here, we use `for` loops to print each block in each row.

- But notice that a pyramid of height 4 is actually a pyramid of height 3, with an extra row of 4 blocks added on. And a pyramid of height 3 is a pyramid of height 2, with an extra row of 3 blocks. A pyramid of height 2 is a pyramid of height 1, with an extra row of 2 blocks. And finally, a pyramid of height 1 is just a pyramid of height 0, or nothing, with another row of a single block added on.
- With this idea in mind, we can write:

```
#include <cs50.h>
#include <stdio.h>

void draw(int h);

int main(void)
{
    // Get height of pyramid
    int height = get_int("Height: ");

    // Draw pyramid
    draw(height);
}

void draw(int h)
{
    // If nothing to draw
    if (h == 0)
    {
        return;
    }

    // Draw pyramid of height h - 1
    draw(h - 1);

    // Draw one more row of width h
    for (int i = 0; i < h; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- Now, our `draw` function first calls itself **recursively**, drawing a pyramid of height `h - 1`. But even before that, we need to stop if `h` is 0, since there won't be anything left to drawn.
- After, we draw the next row, or a row of width `h`.

Merge sort

- We can take the idea of recusion to sorting, with another algorithm called merge sort. The pseudocode might look like:

```
If only one item
  Return
Else
    Sort left half of items
    Sort right half of items
    Merge sorted halves
```

- We'll best see this in practice with an unsorted list:

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

- First, we'll sort the left half (the first four elements):

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

- Well, to sort that, we need to sort the left half of the left half first:

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

- Now, we have just one item, `7`, in the left half, and one item, `4`, in the right half. So we'll merge that together, by taking the smallest item from each list first:

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

- And now we go back to the right half of the left half, and sort it:

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

- Now, both halves of the left half are sorted, so we can merge the two of them together. We look at the start of each list, and take `2` since it's smaller than `4`. Then, we take `4`, since it's now the smallest item at the front of both lists. Then, we take `5`, and finally, `7`, to get:

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

- We now sort the right half the same way. First, the left half of the right half:

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

- Then, the right half of the right half:

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

- We can merge the right half together now:

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

- And finally, we can merge both halves of the whole list, following the same steps as before. Notice that we don't need to check all the elements of each half to find the smallest, since we know that each half is already sorted. Instead, we just take the smallest element of the two at the start of each half:

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

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

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

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

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

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

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

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

- It took a lot of steps, but it actually took fewer steps than the other algorithms we've seen so far. We broke our list in half each time, until we were "sorting" eight lists with one element each:

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

- Since our algorithm divided the problem in half each time, its running time is logarithmic with $O(\log n)$. And after we sorted each half (or half of a half), we needed to merge together all the elements, with n steps since we had to look at each element once.
- So our total running time is $O(n \log n)$:
  - $O(n^2)$
    - bubble sort, selection sort
  - $O(n \log n)$
    - merge sort
  - $O(n)$
    - linear search
  - $O(\log n)$
    - binary search
  - $O(1)$
- Since $\log n$ is greater than 1 but less than n, $n \log n$ is in between n (times 1) and n^2.
- The best case, Ω, is still $n \log n$, since we still sort each half first and then merge them together:
  - $\Omega(n^2)$
    - selection sort
  - $\Omega(n \log n)$
    - merge sort
  - $\Omega(n)$
    - bubble sort
  - $\Omega(\log n)$
  - $\Omega(1)$
    - linear search, binary search
- Finally, there is another notation, Θ, Theta, which we use to describe running times of algorithms if the upper bound and lower bound is the same. For example, merge sort has $\Theta(n \log n)$ since the best and worst case both require the same number of steps.

  And selection sort has $\Theta(n^2)$.

- We look at a final visualization (https://www.youtube.com/watch?v=ZZuD6iUe3Pc) of sorting algorithms with a larger number of inputs, running at the same time.