Lecture 3

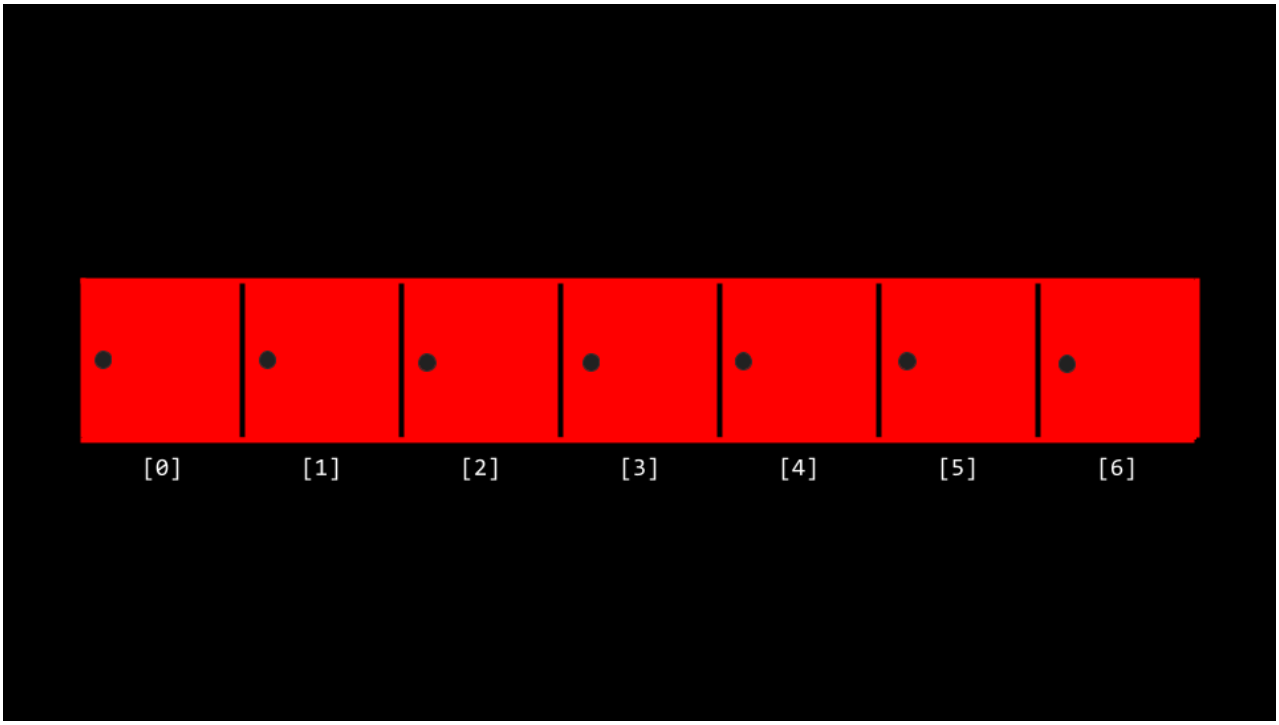
- Welcome!
- Algorithms
- Running Time
- Linear and Binary Search
- Data Structures
- Sorting
- Recursion
- Merge Sort
- Summing Up

Welcome!

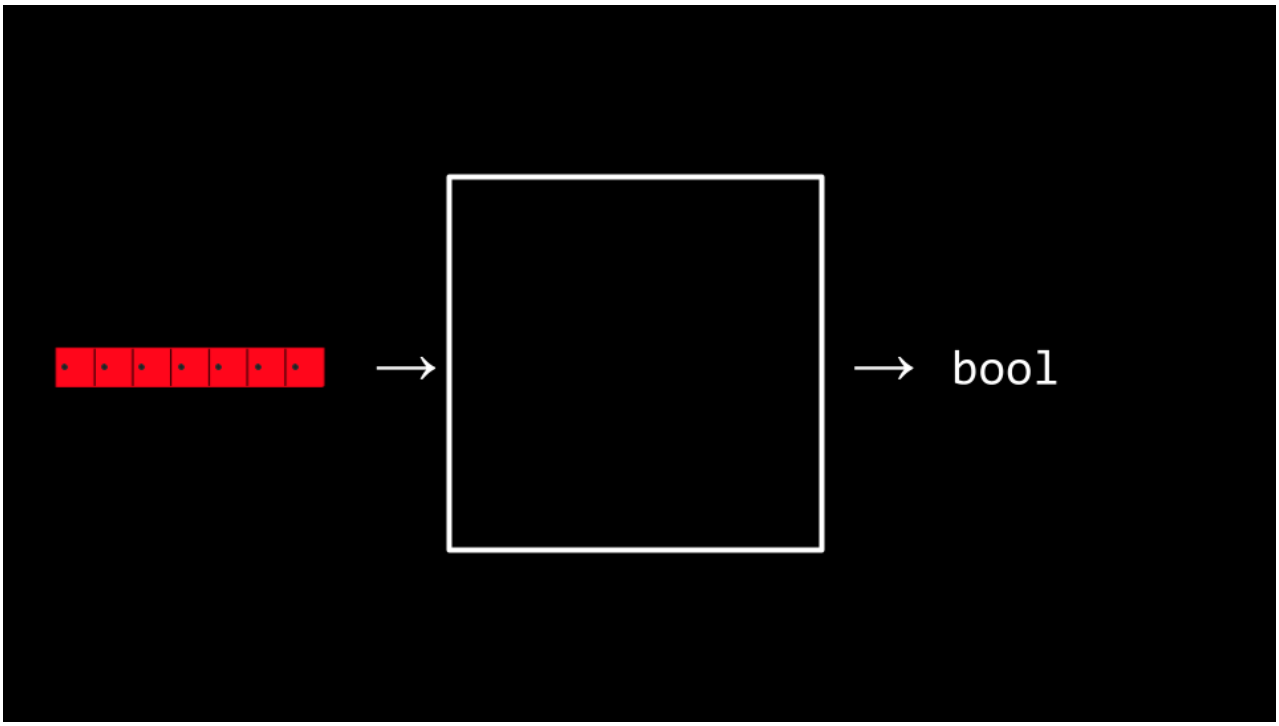
- In week zero, we introduced the idea of an *algorithm*.
- This week, we are going to expand upon our understanding of algorithms through pseudocode and into code itself.
- Also, we are going to consider the efficiency of these algorithms. Indeed, we are going to be building upon our understanding of how to use some of the *lower-level* concepts we discussed last week in building algorithms.

Algorithms

- Recall that last week you were introduced to the idea of an *array*, blocks of memory that are side-by-side with one another.
- You can metaphorically imagine an array like a series of seven red lockers as follows:



- We can imagine that we have an essential problem of wanting to know, "Is the number 50 inside an array?"
- We can potentially hand our array to an algorithm, wherein our algorithm will search through our lockers to see if the number 50 is behind one of the doors: Returning the value true or false.



- We can imagine various instructions we might provide our algorithm to undertake this task as follows:

```
For each door from left to right
    If 50 is behind door
```

```
        Return true
Return false
```

Notice that the above instructions are called *pseudocode*: A human-readable version of the instructions that we could provide the computer.

- A computer scientist could translate that pseudocode as follows:

```
For i from 0 to n-1
    If 50 is behind doors[i]
        Return true
Return false
```

Notice that the above is still not code, but it is a pretty close approximation of what the final code might look like.

- *Binary search* is a *search algorithm* that could be employed in our task of finding the 50.
- Assuming that the values within the lockers have been arranged from smallest to largest, the pseudocode for binary search would appear as follows:

```
If there are no doors
    Return false
If 50 is behind middle door
    Return true
Else if 50 < middle door
    Search left half
Else if 50 > middle door
    Search right half
```

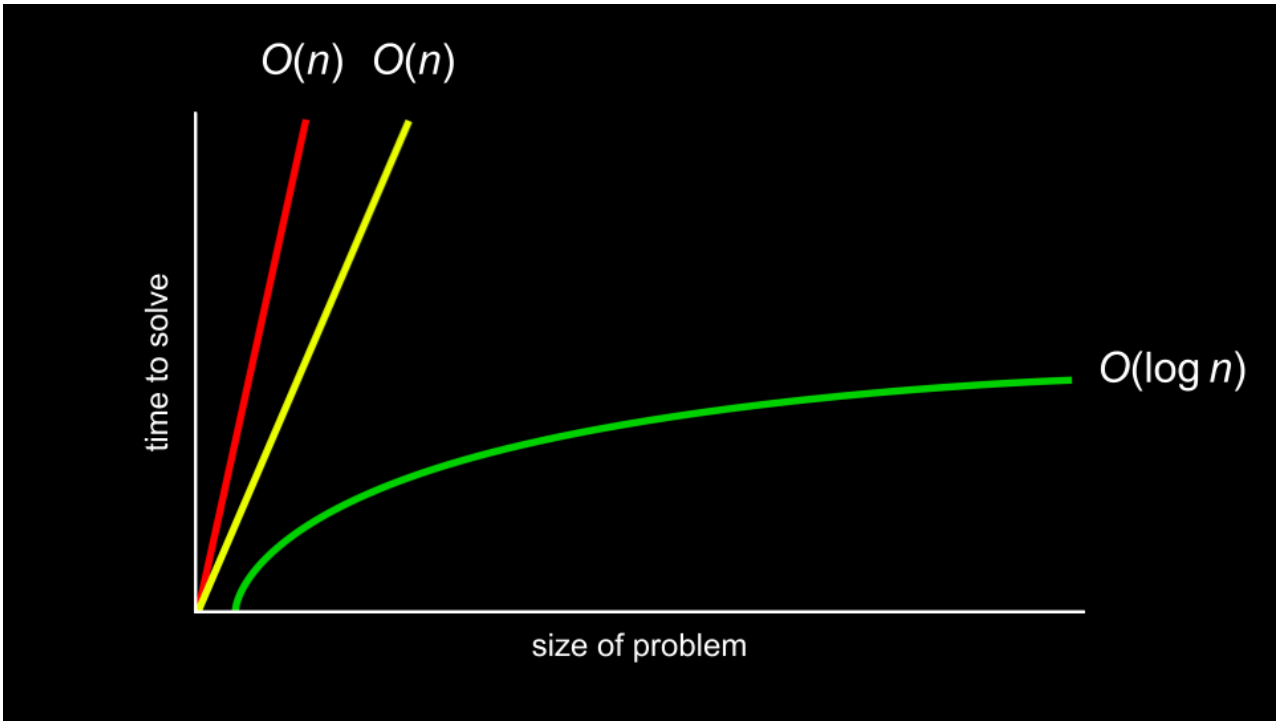
- Using the nomenclature of code, we can further modify our algorithm as follows:

```
If no doors
    Return false
If 50 is behind doors[middle]
    Return true
Else if 50 < doors[middle]
    Search doors[0] through doors[middle-1]
Else if 50 > doors[middle]
    Search doors[middle+1] through doors[n-1]
```

Notice, looking at this approximation of code, you can nearly imagine what this might look like in actual code.

Running Time

- *running time* involves an analysis using *big O* notation. Take a look at the following graph:



- In the above graph, the first algorithm is in $O(n)$. The second is in $O(n)$ as well. The third is in $O(\log n)$.
- It's the shape of the curve that shows the efficiency of an algorithm. Some common running times we may see are:
  - $O(n^2)$
  - $O(n \log n)$
  - $O(n)$
  - $O(\log n)$
  - $O(1)$
- Of the running times above, $O(n^2)$ is considered the worst running time, $O(1)$ is the fastest.
- Linear search was of order $O(n)$ because it could take *n* steps in the worst case to run.
- Binary search was of order $O(\log n)$ because it would take fewer and fewer steps to run even in the worst case.
- Programmers are interested in both the worst case, or *upper bound*, and the best case, or *lower bound*.
- The Ω symbol is used to denote the best case of an algorithm, such as $\Omega(\log n)$.
- The Θ symbol is used to denote where the upper bound and lower bound are the same, where the best case and the worst case running times are the same.

Linear and Binary Search

- You can implement linear search ourselves by typing `code search.c` in your terminal window and by writing code as follows:

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

int main(void)
{
    // An array of integers
    int numbers[] = {20, 500, 10, 5, 100, 1, 50};

    // Search for number
    int n = get_int("Number: ");
    for (int i = 0; i < 7; i++)
    {
        if (numbers[i] == n)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  Notice that the line beginning with `int numbers[]` allows us to define the values of each element of the array as we create it. Then, in the `for` loop, we have an implementation of linear search.

- We have now implemented linear search ourselves in C!
- What if we wanted to search for a string within an array? Modify your code as follows:

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

int main(void)
{
    // An array of strings
    string strings[] = {"battleship", "boot", "cannon", "iron", "thimble", "top h

    // Search for string
    string s = get_string("String: ");
    for (int i = 0; i < 6; i++)
    {
        if (strcmp(strings[i], s) == 0)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
```

```
    return 1;
}
```

Notice that we cannot utilize `==` as in our previous iteration of this program. Instead, we have to use `strcmp`, which comes from the `string.h` library.

- Indeed, running this code allows us to iterate over this array of strings to see if a certain string was within it. However, if you see a *segmentation fault*, where a part of memory was touched by your program that it should not have access to, do make sure you have `i < 6` noted above instead of `i < 7`.
- We can combine these ideas of both numbers and strings into a single program. Type `code phonebook.c` into your terminal window and write code as follows:

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

int main(void)
{
    // Arrays of strings
    string names[] = {"Carter", "David"};
    string numbers[] = {"+1-617-495-1000", "+1-949-468-2750"};

    // Search for name
    string name = get_string("Name: ");
    for (int i = 0; i < 2; i++)
    {
        if (strcmp(names[i], name) == 0)
        {
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Notice that Carter's number begins with `+1-617` and David's phone number starts with '1-949'. Therefore, `names[0]` is Carter and `numbers[0]` is Carter's number.

- While this code works, there are numerous inefficiencies. Indeed, there is a chance that people's names and numbers may not correspond. Wouldn't be nice if we could create our own data type where we could associate a person with the phone number?

Data Structures

- It turns out that C allows a way by which we can create our own data types via a `struct`. Modify your code as follows:

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

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

int main(void)
{
    person people[2];

    people[0].name = "Carter";
    people[0].number = "+1-617-495-1000";

    people[1].name = "David";
    people[1].number = "+1-949-468-2750";

    // Search for name
    string name = get_string("Name: ");
    for (int i = 0; i < 2; i++)
    {
        if (strcmp(people[i].name, name) == 0)
        {
            printf("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Notice that the code begins with `typedef struct` where a new datatype called `person` is defined. Inside a `person` is a string called `name` and a `string` called number. In the `main` function, begin by creating an array called `people` that is of type `person` that is a size of 2. Then, we update the names and phone numbers of the two people in our `people` array. Most important, notice how the *dot notation* such as `people[0].name` allows us to access the `person` at the 0th location and assign that individual a name.

Sorting

- *sorting* is the act of taking an unsorted list of values and transforming this list into a sorted one.
- When a list is sorted, searching that list is far less taxing on the computer. Recall that we can use binary search on a sorted list, but not on an unsorted one.

- It turns out that there are many different types of sort algorithms.
- *Selection sort* is one such search algorithm.
- The algorithm for selection sort in pseudocode is:

```
For i from 0 to n-1
    Find smallest number between numbers[i] and numbers[n-1]
    Swap smallest number with numbers[i]
```

- Consider the unsorted list as follows:

```
  5 2 7 4 1 6 3 0
  ^
```

- Selection sort will begin by looking for the smallest number in the list and swap that number with our current position in the list. In this case, the zero is located and moved to our current position.

```
  0 | 2 7 4 1 6 3 5
```

- Now, our problem has gotten smaller since we know at least the beginning of our list is sorted. So we can repeat what we did, starting from the second number in the list:

```
  0 | 2 7 4 1 6 3 5
      ^
```

- 1 is the smallest number now, so we'll swap it with the second number. We'll repeat this again ...

```
  0 1 | 7 4 2 6 3 5
        ^
```

- ... and again...

```
  0 1 2 | 4 7 6 3 5
          ^
```

- ... and again...

```
  0 1 2 3 | 7 6 4 5
            ^
```

- ... and again ...

```
  0 1 2 3 4 | 6 7 5
              ^
```

- and so on.
- *Bubble sort* is another sorting algorithm that works by repeatedly swapping elements to "bubble" larger elements to the end.

- The pseudocode for bubble sort is:

```
Repeat n-1 times
For i from 0 to n–2
    If numbers[i] and numbers[i+1] out of order
        Swap them
```

- We'll start with our unsorted list, but this time we'll look at pairs of numbers and swap them if they are out of order:

```
5 2 7 4 1 6 3 0
^ ^
2 5 7 4 1 6 3 0
  ^ ^
2 5 7 4 1 6 3 0
    ^ ^
2 5 4 7 1 6 3 0
      ^ ^
2 5 4 1 7 6 3 0
        ^ ^
2 5 4 1 6 7 3 0
          ^ ^
2 5 4 1 6 3 7 0
            ^ ^
2 5 4 1 6 3 0 7
```

- Now, the highest number is all the way to the right, so we've improved our problem. We'll repeat this again:

```
2 5 4 1 6 3 0 | 7
^ ^
2 5 4 1 6 3 0 | 7
  ^ ^
2 4 5 1 6 3 0 | 7
    ^ ^
2 4 1 5 6 3 0 | 7
      ^ ^
2 4 1 5 6 3 0 | 7
        ^ ^
2 4 1 5 3 6 0 | 7
          ^ ^
2 4 1 5 3 0 6 | 7
```

- Now the two biggest values are on the right. We'll repeat again:

```
  2 4 1 5 3 0 | 6 7
  ^ ^
  2 4 1 5 3 0 | 6 7
    ^ ^
  2 1 4 5 3 0 | 6 7
      ^ ^
  2 1 4 5 3 0 | 6 7
        ^ ^
```

```
2 1 4 3 5 0 | 6 7
        ^ ^
2 1 4 3 0 5 | 6 7
```

- … and again …

```
2 1 4 3 0 | 5 6 7
^ ^
1 2 4 3 0 | 5 6 7
  ^ ^
1 2 3 4 0 | 5 6 7
    ^ ^
1 2 3 4 0 | 5 6 7
      ^ ^
1 2 3 0 4 | 5 6 7
```

- … and again …

```
1 2 3 0 | 4 5 6 7
^ ^
1 2 3 0 | 4 5 6 7
  ^ ^
1 2 3 0 | 4 5 6 7
    ^ ^
1 2 0 3 | 4 5 6 7
```

- … and again …

```
1 2 0 | 3 4 5 6 7
^ ^
1 2 0 | 3 4 5 6 7
  ^ ^
1 0 2 | 3 4 5 6 7
```

- … and finally …

```
1 0 | 2 3 4 5 6 7
^ ^
0 1 | 2 3 4 5 6 7
```

- Notice that, as we go through our list, we know more and more of it becomes sorted, so we only need to look at the pairs of numbers that haven't been sorted yet.
- Analyzing selection sort, we made only seven comparisons. Representing this mathematically, where *n* represents the number of cases, it could be said that selection sort can be analyzed as:

```
(n-1)+(n-2)+(n-3)+ ... + 1
```

or, more simply $n^2/2 - n/2$.

- Considering that mathematical analysis, n^2 is really the most influential factor in determining the efficiency of this algorithm. Therefore, selection sort is considered to be of

the order of $O(n^2)$ in the worst case where all values are unsorted. Even when all values are sorted, it will take the same number of steps. Therefore, the best case can be noted as $\Omega(n^2)$. Since both the upper bound and lower bound cases are the same, the efficiency of this algorithm as a whole can be regarded as $\Theta(n^2)$.

- Analyzing bubble sort, the worst case is $O(n^2)$. The best case is $\Omega(n)$.
- You can visualize (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) a comparison of these algorithms.

Recursion

- How could we improve our efficiency in our sorting?
- *Recursion* is a concept within programming where a function calls itself. We saw this earlier when we saw...

```
If no doors
    Return false
If number behind middle door
    Return true
Else if number < middle door
    Search left half
Else if number > middle door
    Search right half
```

Notice that we are calling `search` on smaller and smaller iterations of this problem.

- Similarly, in our pseudocode for Week 0, you can see where recursion was implemented:

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

- Consider how in Week 1 we wanted to create a pyramid structure as follows:

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

- To implement this using recursion, type `code recursion.c` into your terminal window and write code as follows:

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

void draw(int n);

int main(void)
{
    draw(1);
}

void draw(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("#");
    }
    printf("\n");

    draw(n + 1);
}
```

  Notice that the draw function calls itself. Further, note that your code may get caught in an infinite loop. To break from this loop, if you get stuck, hit `ctrl-c` on your keyboard. The reason this creates an infinite loop is that there is nothing telling the program to end. There is no case where the program is done.

- We can correct our code as follows:

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

void draw(int n);

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

    // Draw pyramid
    draw(height);
}

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

```
    // Draw pyramid of height n - 1
    draw(n - 1);

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

Notice the *base case* will ensure the code does not run forever. The line `if (n <= 0)` terminates the recursion because the problem has been solved. Every time `draw` calls itself, it calls itself by `n-1`. At some point, `n-1` will equal `0`, resulting in the `draw` function returning and the program will end.

Merge Sort

- We can now leverage recursion in our quest for a more efficient sort algorithm and implement what is called *merge sort*, a very efficient sort algorithm.
- The pseudocode for merge sort is quite short:

```
If only one number
    Quit
Else
    Sort left half of number
    Sort right half of number
    Merge sorted halves
```

- Consider the following list of numbers:

```
7254
```

- First, merge sort asks, "is this one number?" The answer is "no," so the algorithm continues.

```
7254
```

- Second, merge sort will now split the numbers down the middle (or as close as it can get) and sort the left half of numbers.

```
72|54
```

- Third, merge sort would look at these numbers on the left and ask, "is this one number?" Since the answer is no, it would then split the numbers on the left down the middle.

```
7|2
```

- Fourth, merge sort will again ask , “is this one number?” The answer is yes this time! Therefore, it will quit this task and return to the last task it was running at this point:

```
72|54
```

- Fifth, merge sort will sort the numbers on the left.

```
27|54
```

- Now, we return to where we left off in the pseudocode now that the left side has been sorted. A similar process of steps 3-5 will occur with the right-hand numbers. This will result in:

```
27|45
```

- Both halves are now sorted. Finally, the algorithm will merge both sides. It will look at the first number on the left and the first number on the right. It will put the smaller number first, then the second smallest. The algorithm will repeat this for all numbers, resulting in:

```
2457
```

- Merge sort is complete, and the program quits.
- Merge sort is a very efficient sort algorithm with a worst case of $O(n \log n)$. The best case is still $\Omega(n \log n)$ because the algorithm still must visit each place in the list. Therefore, merge sort is also $\Theta(n \log n)$ since the best case and worst case are the same.
- A final visualization (https://www.youtube.com/watch?v=ZZuD6iUe3Pc) was shared.

Summing Up

In this lesson, you learned about algorithmic thinking and building your own data types. Specifically, you learned…

- Algorithms.
- Big *O* notation.
- Binary search and linear search.
- Various sort algorithms, including bubble sort, selection sort, and merge sort.
- Recursion.

See you next time!