

# CS50: Week-03 - Algorithms

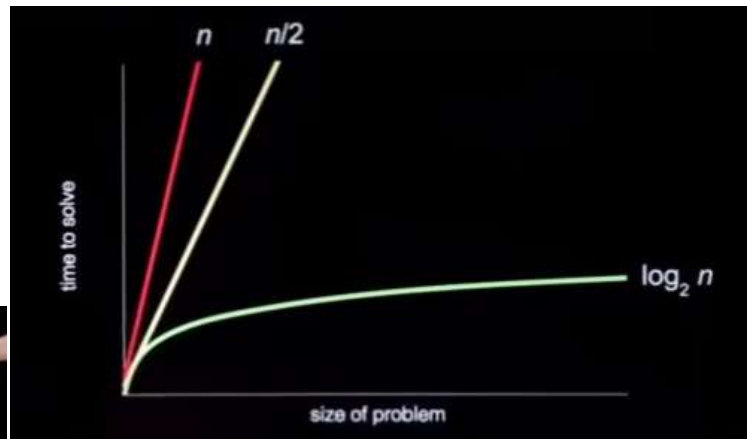
## Lecture

```

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

$n$  times

Binary:  $n/2$

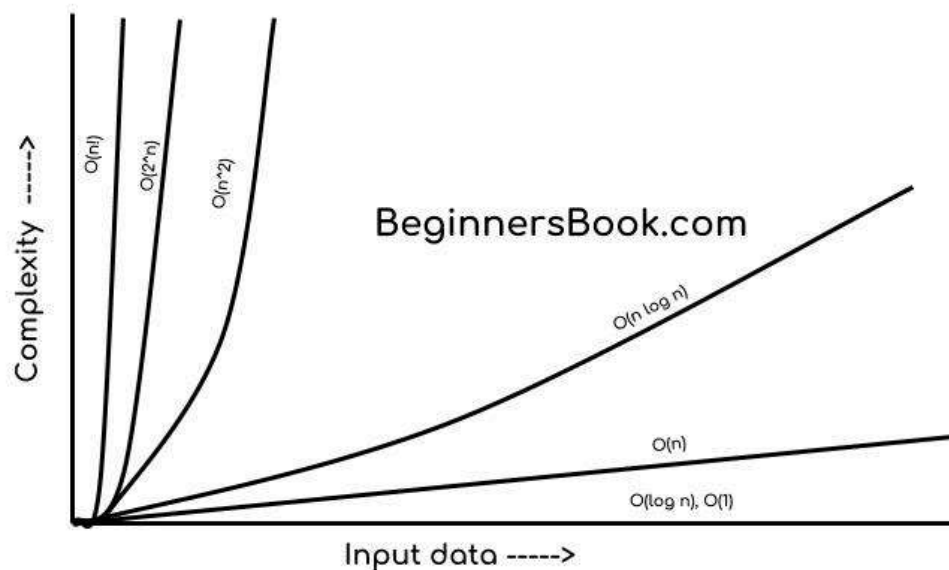


```

If no doors left
    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
    
```

```

If no doors left
    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]
    
```



```

1 #include <cs50.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int numbers[] = {20, 500, -10, 5, 100, 1, -50};
7
8     int n = get_int("Number: ");
9     for (int i = 0; i < 7; i++)
10    {
11        if (numbers[i] == n)
12        {
13            printf("Found\n");
14            return 0;
15        }
16    }
17    printf("Not found\n");
18    return 1;
19 }
20

```

```

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

int main(void)
{
    int numbers[] = {20, 500, -10, 5, 100, 1, -50};
    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;
}

```

In C programming, return 0; at the end of the main() function signifies that the program has executed successfully without any errors.

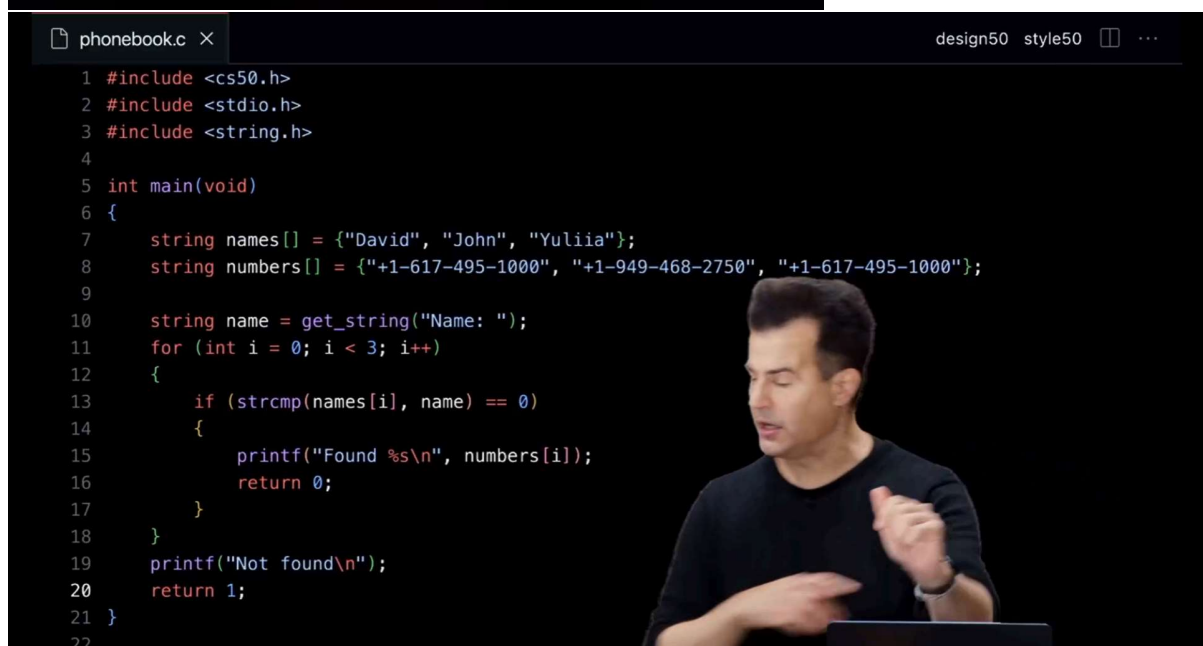
Here's a breakdown:

- **return statement:** This statement terminates the execution of a function and returns control to the calling function.
- **0 value:** By convention, a return value of 0 indicates successful execution.
- **Operating System:** The operating system can use the return value to determine the exit status of the program. This can be useful for scripting and automation purposes.

In C, return 1; signifies that the function is exiting and returning the value 1 to the calling function or operating system.

## Data Structures

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



```
phonebook.c × design50 style50 ...
1 #include <cs50.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     string names[] = {"David", "John", "Yuliia"};
8     string numbers[] = {"+1-617-495-1000", "+1-949-468-2750", "+1-617-495-1000"};
9
10    string name = get_string("Name: ");
11    for (int i = 0; i < 3; i++)
12    {
13        if (strcmp(names[i], name) == 0)
14        {
15            printf("Found %s\n", numbers[i]);
16            return 0;
17        }
18    }
19    printf("Not found\n");
20    return 1;
21 }
22
```

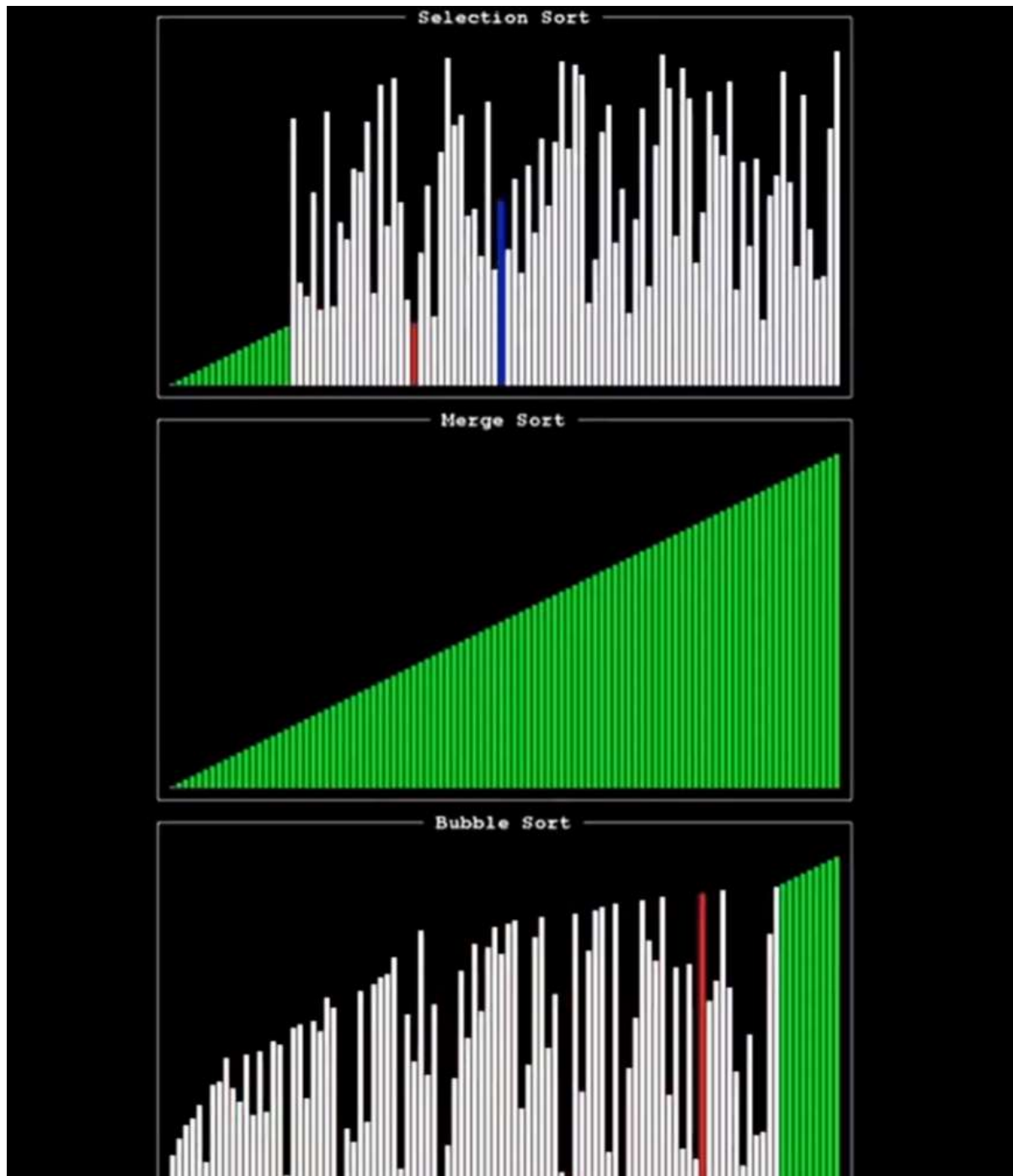
In C, the dot operator ( . ) is used to access members of a structure or union. It's also known as the direct member access operator.

```

11 int main(void)
19     people[1].number = "+1-949-468-2750";
20
21     people[2].name = "Yuliia";
22     people[2].number = "+1-617-495-1000";
23
24     string name = get_string("Name: ");
25     for (int i = 0; i < 3; i++)
26     {
27         if (strcmp(people[i].name, name) == 0)
28         {
29             printf("Found %s\n", people[i].number);
30             return 0;
31         }
32     }
33     printf("Not found\n");
34     return 1;
35 }
36

```





### Section-3

# Recursion

## Factorial

$$1! = 1$$

$$2! = 1 * 2$$

$$3! = 1 * 2 * 3$$

$$4! = 1 * 2 * 3 * 4$$

## Factorial

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1$$

## Factorial

$$f(4)$$

$$4 * f(3)$$

$$3 * f(2)$$

$$2 * f(1)$$

$$1 * f(0)$$

## Factorial

$$f(4) \quad \hookrightarrow \quad 24$$
$$4 * 6$$

Shorts: Linear Search

## Linear Search

Target
9

11	23	8	14	30	9	6	17	22	28	25	15	7	10	19
----	----	---	----	----	---	---	----	----	----	----	----	---	----	----

### In pseudocode:

Repeat, starting at the first element:

- If the first element is what you're looking for (the target), stop.
- Otherwise, move to the next element.

## Shorts: Binary Search

### Binary Search

- In binary search, the idea of the algorithm is to divide and conquer, reducing the search area by half each time, trying to find a target number.
  - In order to leverage this power however, our array must first be sorted, else we cannot make assumptions about the array's contents.

### Binary Search

#### In pseudocode:

- Repeat until the (sub)array is of size 0:
  - Calculate the middle point of the current (sub)array.
  - If the target is at the middle, stop.
  - Otherwise, if the target is less than what's at the middle, repeat, changing the end point to be just to the left of the middle.
  - Otherwise, if the target is greater than what's at the middle, repeat, changing the start point to be just to the right of the middle.

## Binary Search

Target	Start	End	Middle
19	8	14	11

6	7	8	9	10	11	14	15	17	19	22	23	25	28	30
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]

### In pseudocode:

Repeat until the (sub)array is of size 0:

Calculate middle point of the current (sub)array.

If the target is at the middle, stop.

Otherwise, if the target is less than what's at the middle, repeat, changing the end point to be just to the left of the middle.

Otherwise, if the target is greater than what's at the middle, repeat, changing the start point to be just to the right of the middle.

## Binary Search

$$O(\log n)$$

$$\Omega(1)$$

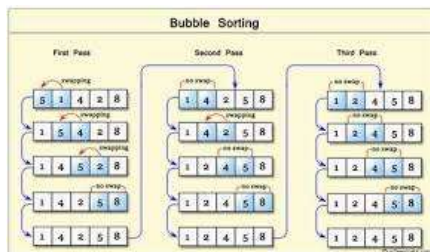
## Bubble Sort

## Bubble Sort

- In bubble sort, the idea of the algorithm is to move higher valued elements generally towards the right and lower value elements generally towards the left.

### In pseudocode:

- Set swap counter to a non-zero value
- Repeat until the swap counter is 0:
  - Reset swap counter to 0
  - Look at each adjacent pair
    - If two adjacent elements are not in order, swap them and add one to the swap counter



## Bubble Sort

Swap Counter
3

2	1	3	5	6	4
---	---	---	---	---	---

### In pseudocode:

Set swap counter to a non-zero value

Repeat until the swap counter is 0:

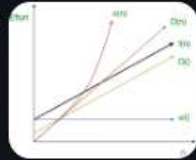
Reset swap counter to 0

Look at each adjacent pair

If two adjacent elements are not in order, swap them and add one to the swap counter



Big O ( $O$ ) and Big Omega ( $\Omega$ ) are notations used to describe the performance of an algorithm. Big O describes the worst-case performance, while Big Omega describes the best-case performance.



## Bubble Sort

- **Worst-case scenario:** The array is in reverse order; we have to “bubble” each of the  $n$  elements all the way across the array, and since we can only fully bubble one element into position per pass, we must do this  $n$  times.
- **Best-case scenario:** The array is already perfectly sorted, and we make no swaps on the first pass.

## Selection Sort

### Selection Sort

- **Worst-case scenario:** We have to iterate over each of the  $n$  elements of the array (to find the smallest unsorted element) and we must repeat this process  $n$  times, since only one element gets sorted on each pass.
- **Best-case scenario:** Exactly the same! There’s no way to guarantee the array is sorted until we go through this process for all the elements.

### Selection Sort

$$O(n^2)$$

$$\Omega(n^2)$$

## Recursion

<https://youtu.be/DA9We62el7E?feature=shared>

## Recursion

- We might describe an implementation of an algorithm as being particularly “elegant” if it solves a problem in a way that is both interesting and easy to visualize.
- The technique of **recursion** is a very common way to implement such an “elegant” solution.
- The definition of a recursive function is one that, as part of its execution, invokes itself.

## Recursion

- The factorial function ( $n!$ ) is defined over all positive integers.
- $n!$  equals all of the positive integers less than or equal to  $n$ , multiplied together.
- Thinking in terms of programming, we’ll define the mathematical function  $n!$  as `fact(n)`.

## Recursion

```
fact(1) = 1
fact(2) = 2 * 1
fact(3) = 3 * 2 * 1
fact(4) = 4 * 3 * 2 * 1
fact(5) = 5 * 4 * 3 * 2 * 1
...
```

## Recursion

- This forms the basis for a **recursive definition** of the factorial function.
- Every recursive function has two cases that could apply, given any input.
  - The *base case*, which when triggered will terminate the recursive process.
  - The *recursive case*, which is where the recursion will actually occur.

## Recursion

### Recursion

```
int fact(int n)
{
    // base case

    // recursive case
}
```

```
int fact(int n)
{
    if (n == 1)
    {
        return 1;
    }

    // recursive case
}
```

### Recursion

```
int fact(int n)
{
    if (n == 1)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}
```

### Recursion

- In general, but not always, recursive functions replace loops in non-recursive functions.

```
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fact(n-1);
}
```

```
int fact2(int n)
{
    int product = 1;
    while(n > 0)
    {
        product *= n;
        n--;
    }
    return product;
}
```

### Recursion

- **Multiple base cases:** The Fibonacci number sequence is defined as follows:
  - The first element is 0.
  - The second element is 1.
  - The  $n^{\text{th}}$  element is the sum of the  $(n-1)^{\text{th}}$  and  $(n-2)^{\text{th}}$  elements.
- **Multiple recursive cases:** The Collatz conjecture.

## Recursion

- The Collatz conjecture is applied to positive integers and speculates that it is always possible to get “back to 1” if you follow these steps:
  - If  $n$  is 1, stop.
  - Otherwise, if  $n$  is even, repeat this process on  $n/2$ .
  - Otherwise, if  $n$  is odd, repeat this process on  $3n + 1$ .
- Write a recursive function `collatz(n)` that calculates how many steps it takes to get to 1 if you start from  $n$  and recurse as indicated above.

## Recursion

n	collatz(n)	Steps
1	0	1
2	1	2 → 1
3	7	3 → 10 → 5 → 16 → 8 → 4 → 2 → 1
4	2	4 → 2 → 1
5	5	5 → 16 → 8 → 4 → 2 → 1
6	8	6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1
7	16	7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1
8	3	8 → 4 → 2 → 1
15	17	15 → 46 → 23 → 70 → ... → 8 → 4 → 2 → 1
27	111	27 → 82 → 41 → 124 → ... → 8 → 4 → 2 → 1
50	24	50 → 25 → 76 → 38 → ... → 8 → 4 → 2 → 1

## Recursion

```
int collatz(int n)
{
    // base case
    if (n == 1)
        return 0;
    // even numbers
    else if ((n % 2) == 0)
        return 1 + collatz(n/2);
    // odd numbers
    else
        return 1 + collatz(3*n + 1);
}
```

## Merge Sort

## Merge Sort

- In merge sort, the idea of the algorithm is to sort smaller arrays and then combine those arrays together (merge them) in sorted order.
- Merge sort leverages something called **recursion**, which we'll touch on in more detail in a future video.

### In pseudocode:

- Sort the left half of the array (assuming  $n > 1$ )
- Sort the right half of the array (assuming  $n > 1$ )
- Merge the two halves together

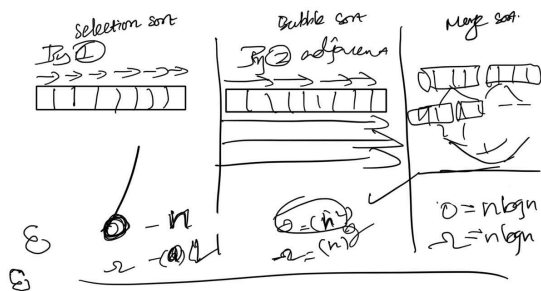
## Merge Sort

- **Worst-case scenario:** We have to split  $n$  elements up and then recombine them, effectively doubling the sorted subarrays as we build them up. (combining sorted 1-element arrays into 2-element arrays, combining sorted 2-element arrays into 4-element arrays...)
- **Best-case scenario:** The array is already perfectly sorted. But we still have to split and recombine it back together with this algorithm.

# Merge Sort

$$O(n \log n)$$

$$\Omega(n \log n)$$



Big O "0" - Worst case scenario. ✓  
 Omega "Ω" - Best case scenario. ✓

