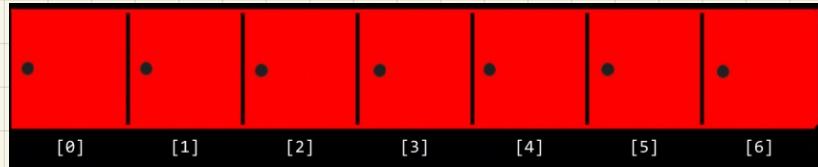




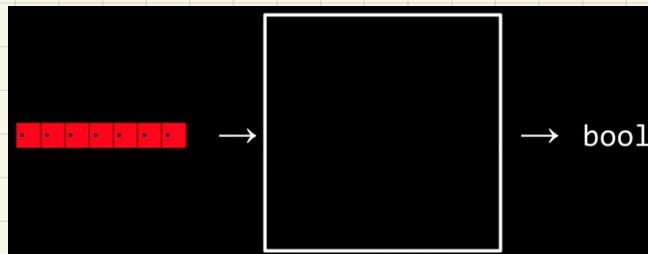
# CS 50-III

## Algorithms

You can metaphorically imagine an array like a series of red lockers.



We can potentially hand our array to an algorithm, wherein our algorithm will search through our lockers to see if the number wanted is behind one of the doors: Returning the value true or false.



We can imagine various instructions we might provide our algorithms to undertake this task:

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

A computer scientist could translate the pseudocode:

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

## Search

### Linear Search

$O(n)$     $\Omega(1)$

In Linear Search, the idea of the algorithm is to iterate across the array from left to right, searching for a specified element.

In pseudocode:

Repeat, starting at the first element:

If the first element is equal to the target, stop  
Else, move to the next element.

You can implement Linear search as follows:

```
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;
}
```

→ `int numbers[7] = {...};` allows us to define the values of each element in the array as we create it.

→ Linear Search

If we are looking for a string, we cannot utilize `==`.

Instead, we have to use `strcmp()`, which comes from the `string.h` library.

⚠ A segmentation fault means a part of the memory was touched by the program that it should not have access to.

## Binary Search

$O(\log n)$

$\Omega(1)$

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 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

In order to leverage this power however, our array must first be sorted, else we can't make assumptions about the array's content.

Assuming that the values within the lockers have been arranged from smallest to largest, the pseudo code Binary search would appear as follows:

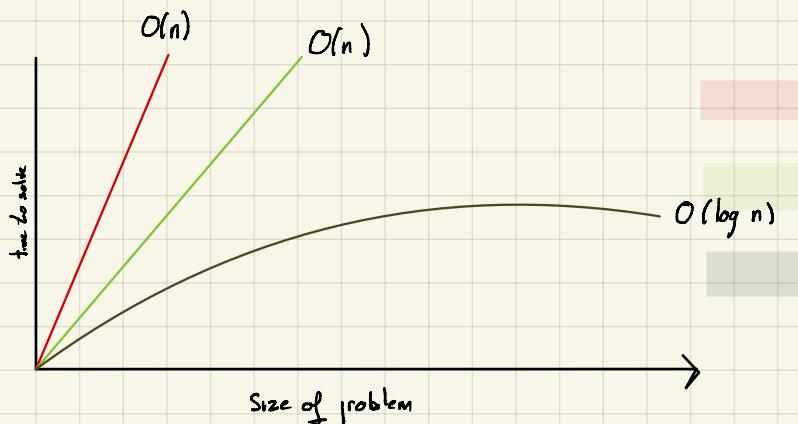
Using the nomenclature of code, we can further modify our algorithm:

```
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]
```

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

## Running Time

Running Time involves an analysis using big O notation.



It's the shape of the curve that shows the efficiency of an algorithm.

Some common running times we may see are:

- $O(n^2)$  ← the worst
- $O(n \log n)$
- $O(n)$
- $O(\log n)$
- $O(1)$  ← the fastest

Linear Search is of order  $O(n)$  because it would take  $n$  steps in the worst case to run.

Binary Search is 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 best case, or upper bound, and the best case, or lower bound.

The  $\Omega$  symbol is used to denote the best case of an algorithm, such as  $\Omega(\log n)$ .

The  $\Theta$  symbol is used to denote where the upper bound and lower bound are the same, where the best and the worst case running times are the same.

## Data Structures

C allows a way by which we can create our own data types via a struct.

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

With `typedef` structure, a new datatype called `person` is defined.

To access the inner variables such as `name` and `number`, we use the dot notation. `person.name` and `person.number`.

# Sorting

Sorting is the fact 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.  
We can use binary search on a sorted list, but not on an unsorted one.

There are many different types of sort algorithms:

## Selection Sort

$O(n^2)$   $\Omega(n^2)$   $\Theta(n^2)$

In Selection Sort, the idea of the algorithm is to find the smallest unsorted element and add it to the end of the sorted list.

In pseudocode:

- Repeat until no unsorted elements remain:
  - Search the unsorted part of the data to find the smallest value.
  - Swap the smallest found value with the first element of the unsorted part.

```
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:

Selection Sort will begin by looking for the smallest value in the list and swap it with our current position. In this case, the 0 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 | 2 7 4 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

$O(n^2)$   $\Omega(n)$

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.

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

Algorithm that works by repeatedly swapping elements to "bubble" larger element to the end.

## Merge Sort

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

In merge sort the idea of the algorithm is to sort smaller arrays and then combine those arrays together (merge) in sorted order. Merge sort leverages recursion.

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.

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

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.

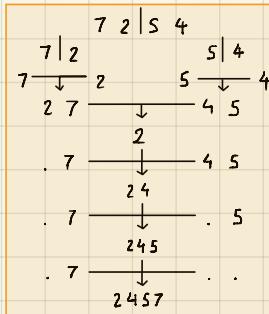
Therefore, merge sort is also  $\Theta(n \log n)$  since the worst case and the best case are the same.

Consider the following list of numbers: 7 2 5 4

First, merge sort checks if this is one number. It's not, so the algorithm continues.

Second merge sort splits the numbers down the middle and sort the left half: 7 2 | 5 4

... and again ... 7 | 2 It is one number. Therefore it will sort and merge. and so on.



## 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.

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 always occur.

It's also possible to have more than one base or recursive cases.

If the program might recurse or terminate in different ways, depending on the input.

### Multiple base cases:

#### Fibonacci number sequence

- 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:

#### Collatz conjecture

- If  $n$  is 1 stop
- Otherwise, if  $n$  is even, repeat the process on  $n/2$ .
- Otherwise, if  $n$  is odd, repeat the process on  $3n+1$ .