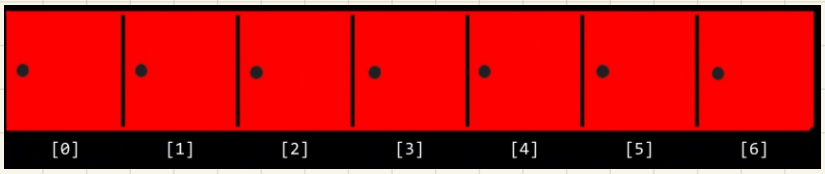
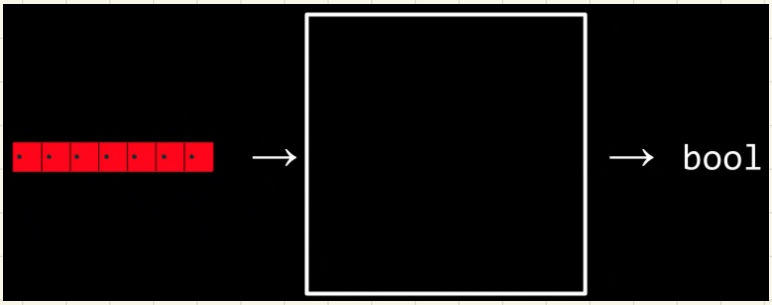
CS50-III

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.



Linear Search

We can imagine various instructions we might provide our algorithm 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
```

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[] = {...} 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

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

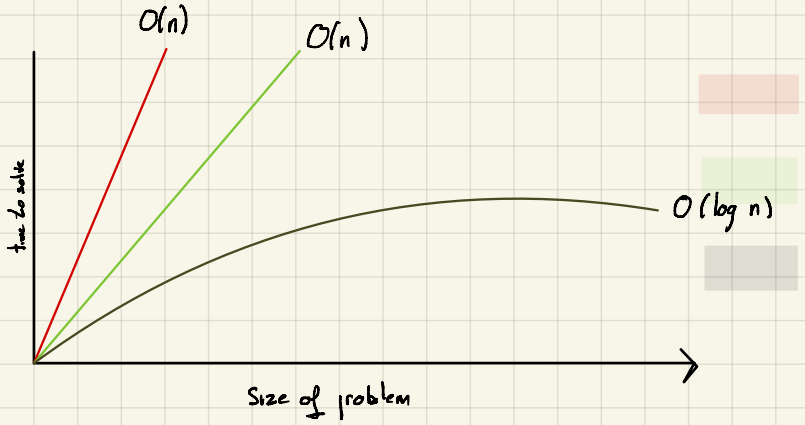
```
If there are no doors
    Return false
If number is behind middle
door
    Re-turn true
Else if number < middle
door
    Search left half
Else if number > middle
door
    Search right half
```

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]
```

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 Ω symbol is used to denote the best case of an algorithm, such as $\Omega(\log n)$.
The Θ symbol is used to denote when 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) Ω (n^2) Θ (n^2)

```
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 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 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 O (n^2) Ω (n^2)

```
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.

Recursion

Recursion is a concept within programming where a function calls itself.

The base case ensures the code does not run forever.

Merge Sort

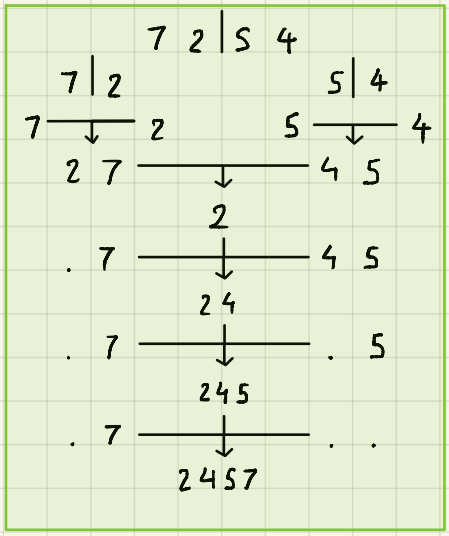
```
If only one number
    Quit
Else
    Sort left half of number
    Sort right half of number
    Merge sorted halves
```

Consider the following list of numbers: 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.



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.