DATA STRUCTURES

Why is understanding data structures relatively important?

Understand what's under the hood of your (JavaScript) car.

Having these tools will help you figure out solutions to many kinds of problems.

Interviews.

A Data structure is a way to store and organize data in a way that makes it efficient to access and manipulate data.

Example: A pile of shoes vs. a shoe rack.

The way that data is stored and organized in a data structure makes certain data structures suitable for doing certain jobs.

Example: A pile of shoes makes putting shoes away easier.

A rack of shoes makes finding shoes easier.

A *graph* to store information about people and their contacts.

A *FIFO queue* to simulate cars waiting for a toll booth.

A fixed length *list* to keep track of CDs in a 6-disc changer.

Data structures can be *linear* like arrays, or *non-linear* such as trees and graphs.

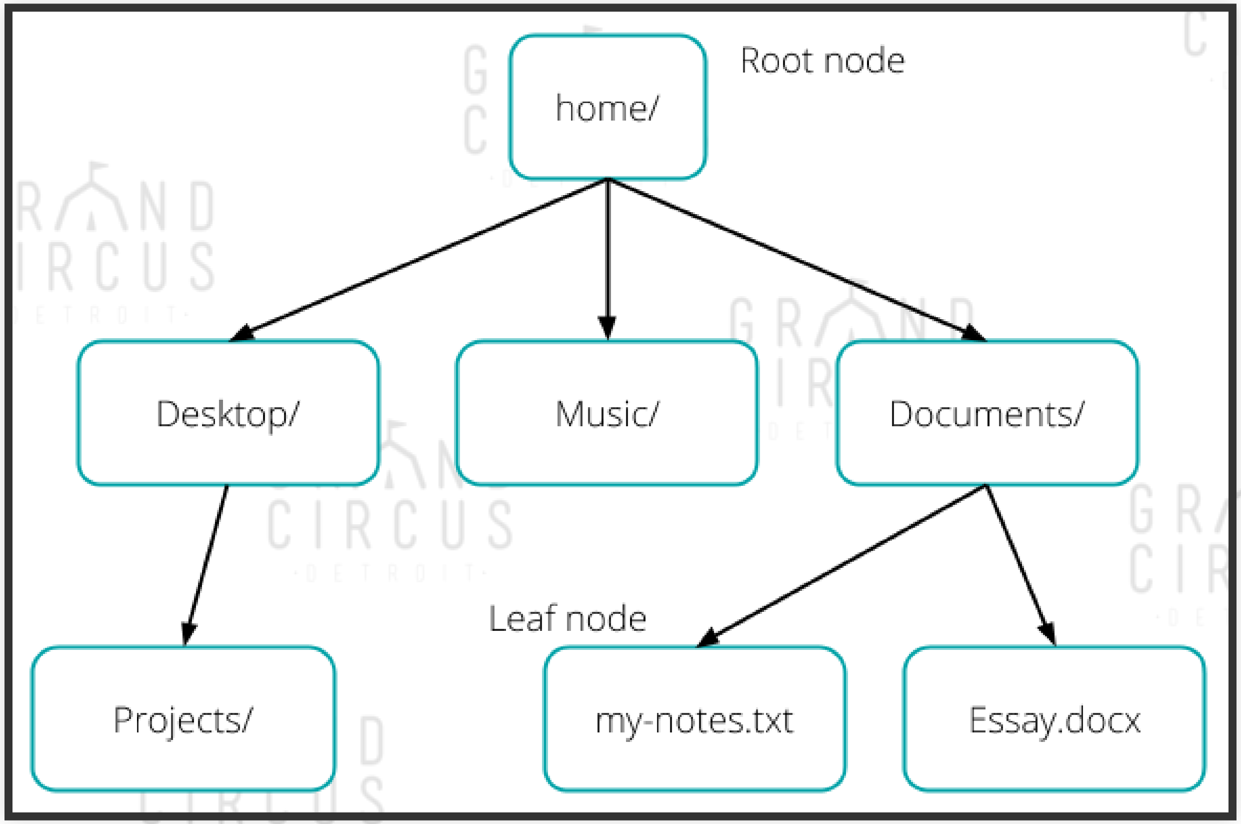
Data structures can have *fixed sized*, or *variable sized* such as JavaScript Arrays.

EXAMPLES OF DATA STRUCTURES

- Arrays
- Trees
- Graphs
- Map
- Stacks
- Queues

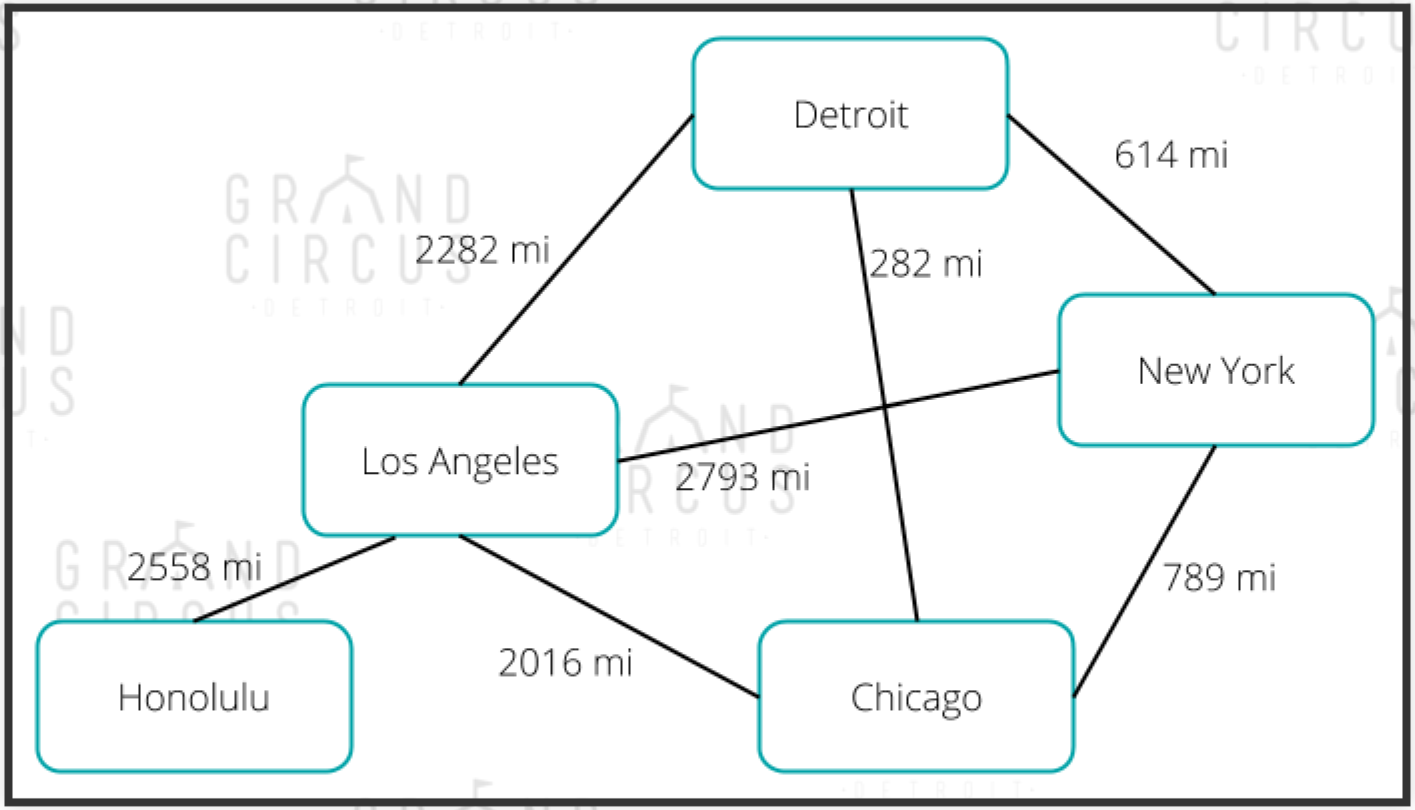
TREE

Nodes are connected in a tree structure, starting at the root and branching out (e.g., a directory structure).



GRAPH

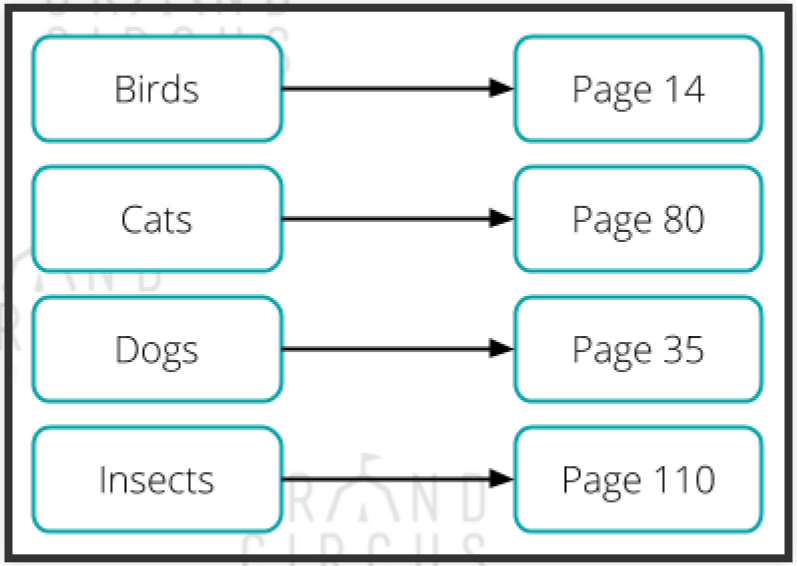
Nodes can be connected in many ways. e.g., routes between cities.



MAP

Values are stored with keys (a.k.a. names).

Great for looking things up quickly (e.g., a topical index in a book).

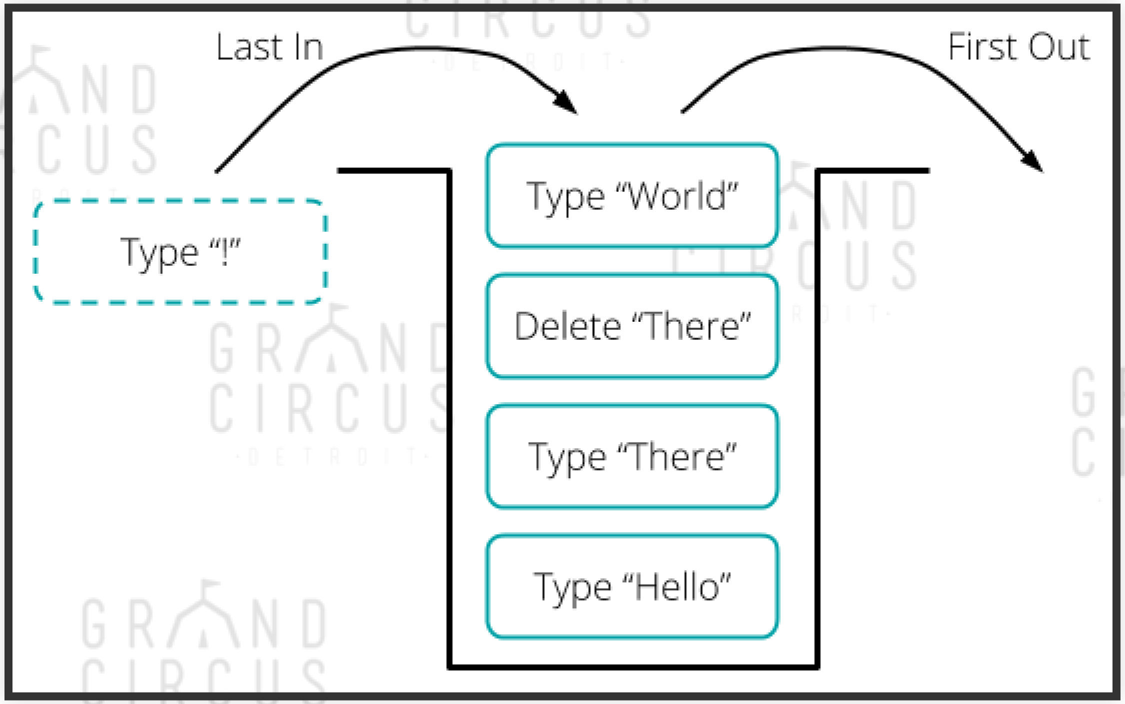


STACK

A list where only one end is accessible.

Also referred to as LIFO (last in - first out).

Imagine an Array with only push and pop (e.g., undo history).

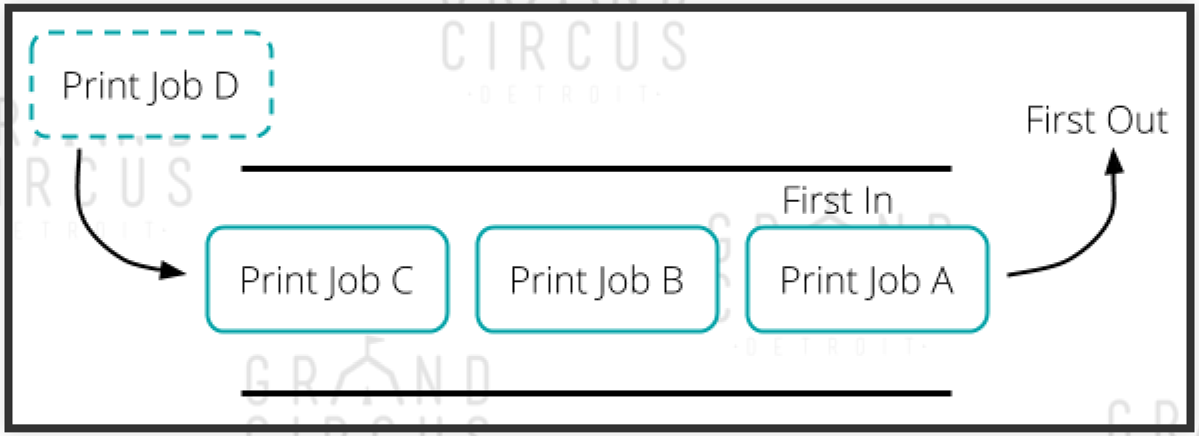


QUEUE

A list where items enter one end and exit the other.

Also referred to as FIFO (first in - first out) (e.g., jobs sent to a printer).

It handles them in the order they came in.



JavaScript has Array and Object data structures built in.

It also has Map and Set.

Map is very much like Objects because it stores a bunch of key-value pairs.

It is more optimized and can have anything as keys, not just strings.

Methods...

- set(key, value)
- get(key)
- has(key)
- delete(key)
- clear()

```
const map = new Map();
map.set("street", "Woodward");
map.set(42, "Answer");
map.set(true, "Yes");
map.has(42); // > true
map.get(true); // > "Yes"
map.size; // 3
map.delete(42);
map; // > Map(2) {"street" => "Woodward", true => "Yes"}
```

Set is a collection that can only have one of each thing.

Methods...

- add(value)
- has(value)
- delete(value)
- clear()

```
const set = new Set();
set.add("Burger");
set.add("Fries");
set.add("Burger"); // NOTE: adding again has no effect
set.size; // > 2
set.has("Burger"); // > true
set; // > Set(2) {"Burger", "Fries"}
```

ABSTRACT DATA TYPES (ADT)

While a *data structure* is the specific way bits and bytes are organized, an *abstract data type* is a defined set of operations you can do with data.

An ADT does not tell you how to structure the data to achieve this purpose.

ABSTRACT DATA TYPES (ADT)

Example: A List must have the following operations

- Push (add to end)
- Pop (remove from end)
- Get (get from a certain index)
- Length
- Clear (remove everything)

There are different ways to store data that would let you do the above five things.

Obviously the array is one of them, but there are others that have various pros and cons.

TWO IMPLEMENTATIONS

- Array List
- Linked List

These are two common data structures that can be used to implement the abstract *List* data type.

TWO IMPLEMENTATIONS

Array List - Implementation of List that is built with a fixed-length array.

Linked List - Implementation of List that is built with a variable-length chain of nodes, where each node links to the next.

The start of the chain is known as the head.

The end of the chain is known as the tail.

UNDER THE HOOD

But before we go any further...

When talking about data structures, it is important to realize JavaScript is a pretty high-level language.

There are several layers between the code we write and what the computer is doing.

UNDER THE HOOD: ARRAY

At the hardware level, variables are stored in memory slots.

More significant values require more memory slots, just like the number 999 requires more digits than 9.

An "array" is a specific number of memory slots that are next to each other.

A longer array requires more slots than a shorter array.

Arrays are very fast and efficient and fundamental to programming, but there's one important limitation: they cannot change size.

You have to tell it how many slots when you make the array.

UNDER THE HOOD: ARRAY

A JavaScript "Array" is very different.

It acts mostly as a memory array, but if you look at the code inside, you will see that it is made with different data structures.

It is less efficient, but it can change size, which is very convenient.

WHY DOES IT MATTER?

We do not usually have to deal with this stuff when writing JavaScript, but like your car, occasionally you do need to dig in when something goes wrong or when you need to improve performance.

Not to mention if you ever write in another programming language…

ONE HAND TIED BEHIND OUR BACK

So for the following exercise, we are going to simulate using a hardware array.

We will still use a JavaScript array because that is what we know, but with these caveats:

1. Arrays have fixed length. They cannot get any longer or smaller.
2. We can only access arrays using `[]` and `.length`. No push, pop, splice, and forEach.

BUILDING A LIST

Now let's dig into building the List.

- Using an Array
- Using Linked Nodes

BUILDING A LIST

Let's go to the whiteboard.

- Print ("traverse" list and console log everything)
- Push (add to end)
- Pop (remove from end)
- Length
- Unshift (add to beginning)
- Clear (remove everything)
- Get (get from a certain index)

EVALUATE

- What was easy or hard when using an array?
- What was easy or hard when using a linked list?

EVALUATE

	Pro (easy)	Con (hard)
Array List	get, push, pop	shift, unshift, (constrained space)
Linked List	shift, unshift	get

ALGORITHMS

WHAT ARE ALGORITHMS?

An algorithm is a finite set of steps that expresses how a specific task should be done.

WHAT ARE ALGORITHMS?

Algorithms do not represent complete programs; they represent the core of the problem.

Algorithms can be expressed using pseudocode or flowcharts.

EXAMPLE ON ALGORITHMS

Design an algorithm to add two numbers, then displaying the result.

EXAMPLE ON ALGORITHMS

Steps:

1. START
2. Declare three integers x,y, and z
3. Define values for x and y.
4. Add values of x and y
5. store output of step 4 into z
6. print z
7. STOP

CHARACTERISTICS OF AN ALGORITHM

- Has Finite set of steps
- Unambiguous: Each step should be clear and means one thing.
- May have 0 or more inputs
- Has one or more outputs
- Feasible: Possible to make into code
- Independent: Can work with any programming language.

SOME TYPES OF ALGORITHMS

- Greedy
- Divide and Conquer

GREEDY ALGORITHMS

These types of algorithms try to find the solution without thinking ahead.

At each point, do the next best thing.

This may or may not be best.

Example: If trying to get to California, you always turn West at every intersection.

GREEDY ALGORITHMS

An example where it does work well: Counting Coins.

1. Initialize result as empty.
2. Find the largest denomination that is smaller than V.
3. Add found denomination to result. Subtract the value of found denomination from V.
4. If V becomes 0, then print result.
5. Else repeat steps 2 and 3 for the new value of V

GREEDY ALGORITHMS

An example where it *sometimes* works well: Knapsack problem.

If you can stuff items of different sizes and values into a knapsack, which items give you the most value and still fit?

A greedy algorithm would always pick the most expensive items.

KNAPSACK PROBLEM

Which items do you grab if your knapsack can carry 25lb? (the optimal answer is A & D)

Item	Weight	Value
A	20 lb	$ 30
B	10 lb	$ 9
C	10 lb	$ 9
D	2 lb	$ 3

KNAPSACK PROBLEM

Which items do you grab if your knapsack can carry 25lb? (the optimal answer is B & C, but greedy picks A)

Item	Weight	Value
A	20 lb	$ 12
B	10 lb	$ 9
C	10 lb	$ 9
D	7 lb	$ 5

DIVIDE AND CONQUER ALGORITHMS

In this approach, a problem is divided into smaller sub-problems, until there can be no more room for smaller sub-problems.

Those sub-problems can be independently solved, then eventually merged to get the solution for the original problem.

Example: *Merge Sort*

DIVIDE AND CONQUER

MERGE SORT

Let's try it!

SEARCH ALGORITHMS

- Linear Search
- Binary Search

LINEAR SEARCH

Linear Search is a straightforward approach to search for an item in a list.

The idea is to scan the whole list, until you find the element, or reach the end of the list.

LINEAR SEARCH

Algorithm: Linear Search (Array A, Value x)

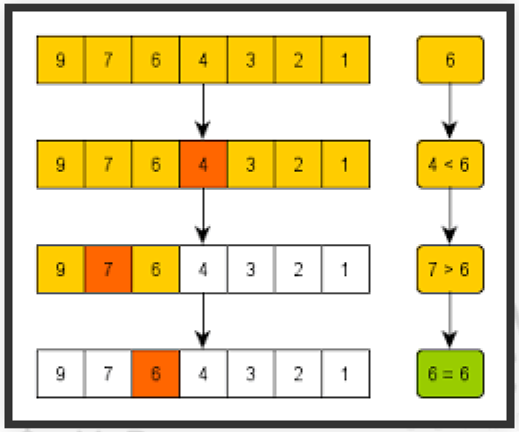
- Step 1: Set i to 1
- Step 2: if i > n then go to step 7
- Step 3: if A[i] = x then go to step 6
- Step 4: Set i to i + 1
- Step 5: Go to Step 2
- Step 6: Print Element x Found at index i and go to step 8
- Step 7: Print element not found
- Step 8: exit

BINARY SEARCH

If you have a sorted list of data, you can keep picking a point in the middle and checking if you need to go higher or lower.

This only works with a sorted data.

BINARY SEARCH



BINARY SEARCH

```
const binarySearchRecursive = (inputArray, key, min, max) => {
  if (min > max) {
    return null; // Nothing left to search. It's not in the array.
  } else {
    const mid = Math.floor((min + max) / 2); // Find the middle
    const element = inputArray[mid];
    if (key === element) {
      return mid; // Found it!
    } else if (key < element) {
      // It's in the first half.
      return binarySearchRecursive(inputArray, key, min, mid - 1);
    } else {
      // It's in the second half.
      return binarySearchRecursive(inputArray, key, mid + 1, max);
    }
  }
};
const arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'];
console.log(binarySearchRecursive(arr, 'G', 0, arr.length - 1));
```

COMPLEXITY ANALYSIS

In Algorithms, studying time/space complexity can tell us how good an Algorithm is.

Here, we define a mathematical representation of its run-time/space performance.

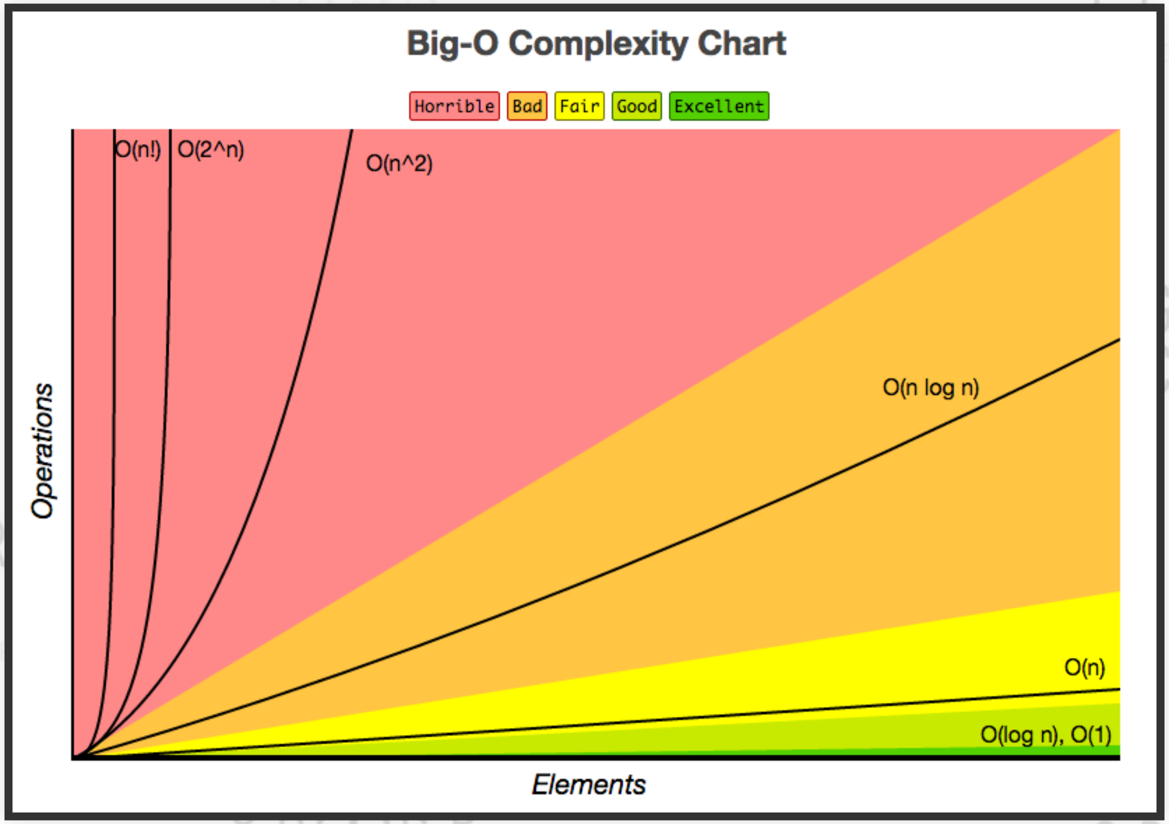
This is often described using *Big O Notation*.

COMPLEXITY ANALYSIS

Algorithm runtime estimations fall into three categories:

- Best Case: Minimum time required to run the algorithm.
- Average Case: Average time required to run the algorithm.
- Worst Case: Maximum time required to run the algorithm.

COMPLEXITY ANALYSIS



EXAMPLES

- Array.pop() / no loop - O(1)
- Binary Search - O(log n)
- Linear Search / simple for loop - O(n)
- Merge Sort - O(n log n)
- Selection Sort / a nested for loop - O(n^2)
- Naive Recursive Fibonacci - O(2^n)

RULES OF THUMB FOR ESTIMATIONS

No Loops	O(1)
Loops, but not nested	O(n)
Nested Loops	O(n^2)
Divide and Conquer	O(log n) or O(n log n)

SORTING

Let's explore some different sort algorithms.

- Insertion
- Selection
- Bubble
- Merge Sort (*space requirement)
- Bucket Sort

There are many good articles and videos on the internet to dig into each of these.