Basic Algorithms Analysis in Javascript

León Jaramillo

softserve

Javascript as a Multiparadigm Language

Imperative	Procedural	Object-Oriented	Event-Driven
Declarative	Functional	Prototype-based	Dynamic typing

Imperative Programming vs. Declarative Programming

Imperative Programming

- It's about how to do things
- We use statements to say to the computer what to do
- It's about manipulating the state of the system via its data structures
- We rely on decision, branching and cycle statements

Declarative Programming

- It's about what to do
- We use expressions, that when evaluated tell the computer what we want
- It's about not changing the state of immutable data structures within the system
- We rely, typically on pure functions

Object-Oriented Programming (OOP) vs. Functional Programming (FP)

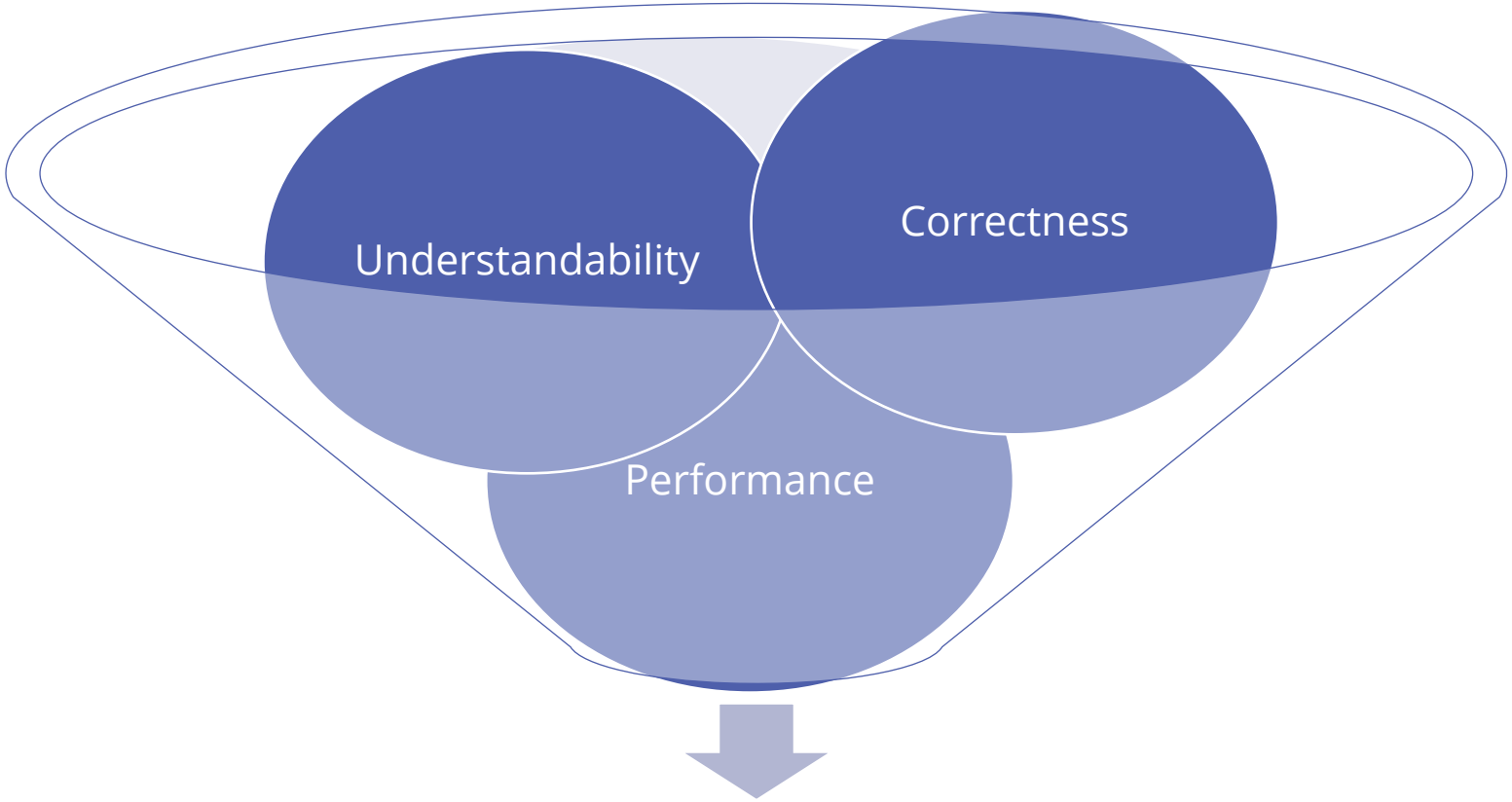
OOP

- Based on abstractions using objects
- Relies on mutable data
- It's an imperative approach
- Tightly related to iterations
- Structured using objects with operations
- Suitable for domains with many things with few operations
- Objects are more reusable than their operations

FP

- Based on evaluating functions
- Relies on immutable data
- It's a declarative approach
- Tightly related to recursion
- Structured using functions which process data structures
- Suitable for domains with few things with many operations
- Functions are highly reusable across the whole domain

Performance as a Key Algorithms Analysis Driver



“Good” algorithm

How to Measure Algorithms Performance in Javascript?

- There are different ways to **measure performance** in WebUI environments (Chrome DevTools, for example).
- There are, also, different approaches to **evaluate** it (the RAIL model, for example).
- But the **simplest way** to measure a JS algorithm's performance is using `performance.now()`.
- It's just about measuring how many **milliseconds** takes to execute the algorithm.
- However, it is **not very reliable**, because its results depend on many external factors.

Algorithms Complexity

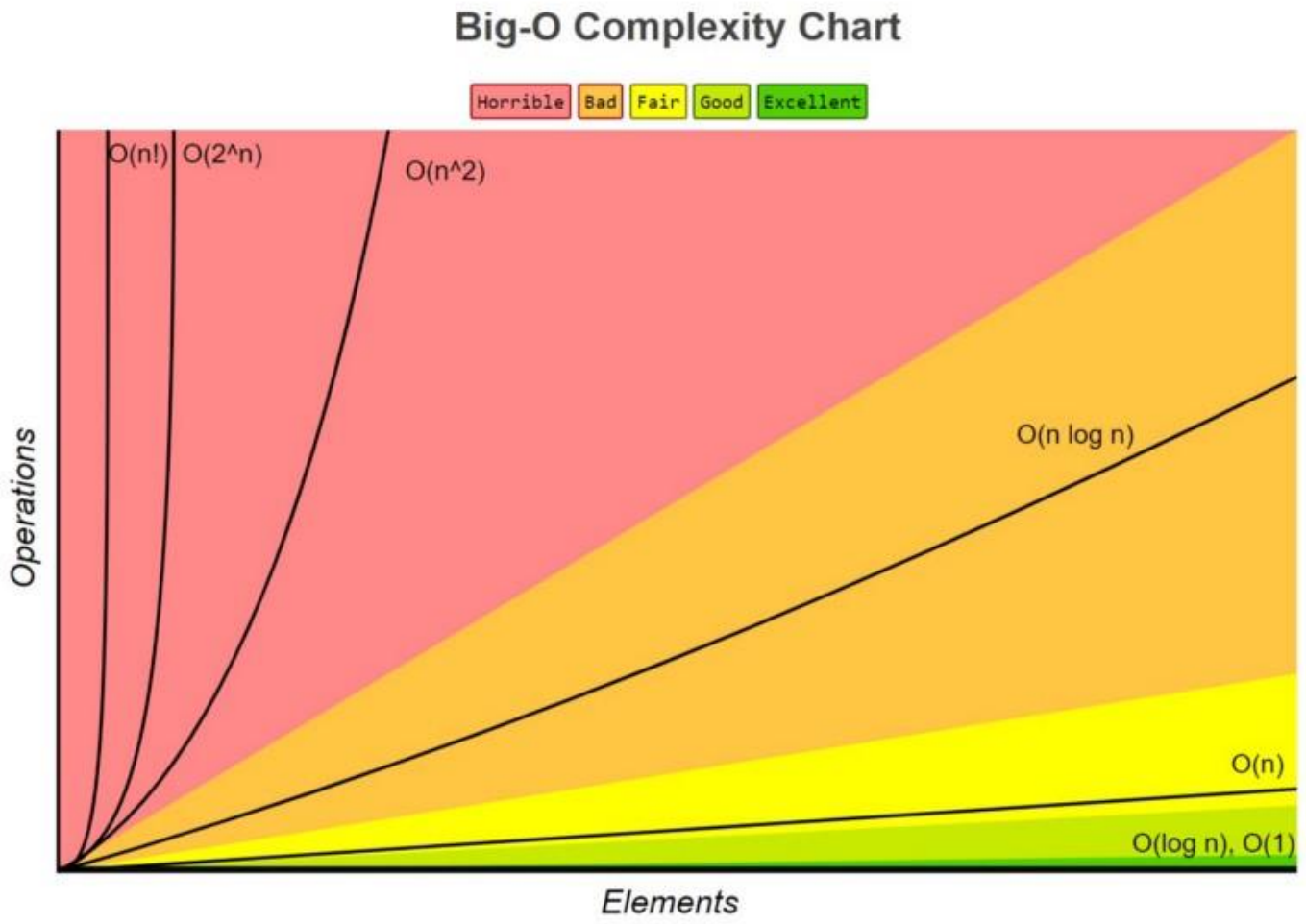
Pure numbers

- Time
- Space

Expressing Time Complexity using Big-O Notation

- **Big-O Notation** is widely used to express the time complexity of an algorithm.
- It's a **mathematical notation** that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
- It's used to **classify** algorithms according to their complexity. The most usual of those classes are:
  - *O(1)*: Constant
  - *O(log n)*: Logarithmic
  - *O(n)*: Linear
  - *O(n log n)*: Loglinear
  - $O(n^2)$: Quadratic
  - $O(2^n)$: Exponential
  - *O(n!)*: Factorial

Expressing Time Complexity using Big-O Notation



See this with well-known algorithms: https://www.bigocheatsheet.com/

softserve

Big-o Notation via Asymptotic Analysis (1/4)

For determining the time complexity of an algorithm using asymptotic analysis, we shall first **determine the Time function** `T` of the algorithm regarding the number `n` of input elements `T(n)`. We'll follow the next steps:

1. We'll assume that executing every statement takes a **similar amount of time** (we could assume arbitrary **constants**).
2. The execution time of one code block is the **sum** of the times of each statement within it.
3. The execution time of an `if…else` statement is, at most, the time of the **longest alternative**.
4. The execution time of a loop is, at most, the sum of the times of each statement within it, multiplied for its **number of iterations**.
5. The execution time of a **nested loop** will result from multiplying the time that takes each of both loops.

Big-o Notation via Asymptotic Analysis (2/4)

Once we know the Time function T(n) of the algorithm, we may find it's Big-O time complexity, with the following steps:

1. Find the **fastest growing** term of the function.

2. **Remove the coefficient** of that term.

3. Put the result **within the parentheses** of the Big-O Notation expression.

Big-o Notation via Asymptotic Analysis (3/4)

```
function sum(n) {
    let sum = 0;
    for (let  i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

1. Determined the Time function

   `T(n) = a + b*n + c`

1. Found the fastest growing term

   `b*n`

1. Remove the coefficient of that term

   `n`

1. Get the Big-O Notation expression

   O(n)

Big-o Notation via Asymptotic Analysis (4/4)

```
function printIdentityMatrix(n) {
    for (let  i = 0; i < n; i++) {
        for (let  j = 0; j < n; j++) {
            if(i==j) {
                document.write('1 ');
            }
            else {
                document.write('0 ');
            }
        }
        document.write('<br>');
    }
}
```

1. Determined the Time function

   `T(n) = n*(a + b*n)`

1. Found the fastest growing term

   `n*(a + b*n)`

1. Remove the coefficient of that term

   `n*n -> ` n^2

1. Get the Big-O Notation expression

   $O(n^2)$

What About Doing these Analyses in Functional Programming?

- Finding the time complexity of a functional algorithm can be **trickier**.
- In functional programming, we **don't see loops** that often.
- Instead, we see **functions** (even pure or high-order ones) that often are **harder to analyze**:
  - `Filter`
  - `Map`
  - `Reduce`

References and Resources

- Imperative vs Declarative programming in JavaScript: https://medium.com/weekly-webtips/imperative-vs-declarative-programming-in-javascript-25511b90cdb7
- Functional Programming VS Object Oriented Programming (OOP) Which is better....? https://medium.com/@shaistha24/functional-programming-vs-object-oriented-programming-oop-which-is-better-82172e53a526
- performance.now(): https://developer.mozilla.org/es/docs/Web/API/Performance/now
- Big-O Cheatsheet: https://www.bigocheatsheet.com/
- Algorithms design with JavaScript: https://www.youtube.com/watch?v=JgWm6sQwS_I
- JavaScript Algorithms and Data Structures: https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/
- Big-O Notation Calculator: https://shunnarski.github.io/BigO.html

softserve

FOR THE FUTURE

Thanks! Any question?