Introduction to Big-O Notation

Goals

Develop a conceptual understanding of Big-O notation

- Explain need for notation
- Analyze time complexity
- Compare different time complexities

Big-O Notation

What's the idea here?

- Imagine we have multiple implementations of the same function
- How can we determine which one is the "best?"
- *Function that accepts a string and returns reversed copy*
  - Good?
  - Bad?
  - Meh?

Who cares?

- It's important to have precise vocabulary about how code performs
- Useful for discussing trade-offs between different approaches
- When code slows, identifying inefficient parts helps find pain points
- Less important: it comes up in interviews!

An example

- Calculate sum of numbers from 1 up to *(and including)* some number ***n***

```
function addUpToFirst(n) {

  let total = 0;

  for (let i = 1; i <= n; i++) {
    total += i;
  }

  return total;
}
```

```
function addUpToSecond(n) {
  return n * (n + 1) / 2;
}
```

Which is better?

What does better mean?

- Faster?
- Less memory-intensive?
- More readable?
- Let's focus on *speed*
- We can time them! <https://rithmschool.github.io/function-timer-demo/>

The problem with timers

- Different machines will record different times
- The same machine will record different times!
- For fast algorithms, speed measurements may not be precise enough
- Instead, count number of simple operations the computer has to perform!

If not time, then what?

- Rather than counting *seconds*, which are so variable…
- Let's count *number* of simple operations the computer has to perform!

Let's try counting operations!

```
function addUpToSecond(n) {
  return n * (n + 1) / 2;
}
```

3 simple operations, regardless of the size of ***n***

Another example

```
function addUpToFirst(n) {

  let total = 0;

  for (let i = 1; i <= n; i++) {
    total += i;
  }

  return total;
}
```

Let's try counting number of operations!

What have we learned?

- Counting is hard!
- Regardless of exact number, number of operations grows proportional to ***n***
  - If ***n*** doubles, number of operations will also double
- We can use this idea to measure speed allocation of algorithms

Introducing... Big O

- Big O Notation is a way to formalize fuzzy counting
    - Can use to talk about how the runtime of algorithm grows as inputs grow
- We won't care about the details, only the trends

Big O Definition

An algorithm is ***O(f(n))*** if number of simple operations is eventually less than a constant times ***f(n)***, as ***n*** increases

- f(n) could be linear (f(n) = n)
- f(n) could be quadratic (f(n) = n^2)
- f(n) could be constant (f(n) = 1)
- f(n) could be something entirely different!

Back to our example

```
function addUpToSecond(n) {
  return n * (n + 1) / 2;
}
```

- Always 3 operations
- O(1)

```
function addUpToFirst(n) {

  let total = 0;

  for (let i = 1; i <= n; i++) {
    total += i;
  }

  return total;
}
```

- The number of operations is bounded by a multiple of ***n*** (say, 10n)
- This algorithm "runs in" O(n)

Another example

```
function printAllPairs(n) {
  for (var i = 0; i < n; i++) {
    for (var j = 0; j < n; j++) {
      console.log(i, j);
    }
  }
}
```

- O(n) operation inside of an O(n) operation
- This algorithm "runs in" O(n^2)

Worst Case

Big O notation is concerned with *worst case* of algorithm's performance.

```
function allEven(nums) {
  for (var i = 0; i < nums.length; i++) {
    if (nums[i] % 2 !== 0) {
      return false;
    }
  }
  return true;
}
```

This is O(n), since even though it may not always take ***n*** times, it will scale with ***n***

Simplifying Big O Expressions

- When determining algorithm time complexity, rule for big O expressions:
  - Constants **do not** matter
  - Smaller terms **do not** matter
  - Always make sure you can answer - **what is *n***?

Helpful hints

- Arithmetic operations are constant
- Variable assignment is constant
- Accessing elements in array *(by index)* or object *(by key)* is constant
- Loops: length of the loop times complexity of whatever happens in loop

Common Runtimes

log what?

- We're in base 2 (think about 0s and 1s)
- $\log_2 8$ = 3 (2 to the power of what gives me 8?)
- The logarithm of a number roughly measures the number of times you can divide that number by 2 before you get a value that's less than or equal to one.
- Logarithmic time complexity is great! You've written an algorithm that can find a value in a sorted array in $\log_2 n$ time!

What's the difference?

For ***n*** = 100:

Type	Function	Result
Constant	1	1
Logarithmic	log n	≈7
Linear	n	100
Logarithmic	n log n	≈664
Quadratic	n^2	10,000

Type	Function	Result
Exponential	2^n	1,267,650,600,228,229,401,496,703,205,376
Factorial	n*!*	$\approx 9.332622 \times 10^{157}$

How about things we know?

- What is the time complexity of ***.includes()***?
- What is the time complexity of ***.indexOf()***?

Must knows for now

- A loop does not mean it's O(n)!
- A loop in a loop does not mean it's $O(n^2)$!
- Best runtime for sorting is $O(n \times \log_2 n)$ (also referred to as $n \log_2 n$)
- It is *not* same as $\log_2 n$

Space Complexity

So far, we've been focusing on **time complexity**: how can we analyze runtime of an algorithm as size of inputs increase?

Can also use big O notation to analyze **space complexity**: how will memory usage scale as size of inputs increase?

Rules of Thumb in JS

- Most primitives (booleans, numbers, ***undefined***, ***null***): constant space
- Strings: O(n) space (where ***n*** is the string length)
- Reference types: generally O(n), where ***n*** is the length of array (or keys in object)

An example

```
function sum(nums) {

  let total = 0;

  for (let i = 0; i < nums.length; i++) {
    total += nums[i];
  }

  return total;
}
```

O(1) space

Another example

```
function double(nums) {

  let doubledNums = [];

  for (let i = 0; i < nums.length; i++) {
    doubledNums.push(2 * nums[i]);
  }

  return doubledNums;
}
```

O(n) space

- Time complexity is more of the focus for now
- We will be covering space complexity in more detail later on in the course

Recap

- To analyze performance of algorithm, use Big O Notation
  - Can give high level understanding of time or space complexity
  - Doesn’t care about precision, only general trends (linear? quadratic? constant?)
  - Depends only on algorithm, not hardware used to run code
- Big O Notation is everywhere, so get lots of practice!