NodeJS

LABA

BIG O NOTATION

Solvd NodeJS LABA

WHAT IS BIG O

Big O is a way to categorize your algorithms time or memory requirements based on input. It is not meant to be an exact measurement. It will not tell you how many CPU cycles it takes, instead, it is meant to generalize the growth of your algorithm.

So when someone says “O of N”, they say your algorithm will grow linearly with the input. Think of a traversing a list

WHY DO WE USE IT

Often it will help us make decisions about what data structures and algorithms to use. Knowing how they will perform can greatly help create the best possible program to solve a problem.

Let’s do an small example, look at the following code, what is the running time of that function?

BIG O NOTATION

```
function sumCharCodes(needle) {
  let sum = 0;
  for (let i = 0; i < needle.length; i++) {
    sum += needle.charCodeAt(i);
  }

  return sum;
}
```

Solvd
NodeJS LABA

BIG O NOTATION

Big O, said differently is like asking the question: “as you input grows, how fast does computation or memory grows?”

In the real world, memory is not free. But we don’t really consider that when thinking about algorithms.

Let’s go back to our example, how does our program’s execution time grows in regards to input?

BIG O NOTATION

```
function sumCharCodes(needle) {
  let sum = 0;
  for (let i = 0; i < needle.length; i++) {
    sum += needle.charCodeAt(i);
  }

  return sum;
}
```

BIG O NOTATION

```
function sumCharCodes(needle) {
  let sum = 0;
  for (let i = 0; i < needle.length; i++) {
    sum += needle.charCodeAt(i);
  }

  return sum;
}
```

We have a N relationship. O(N) time complexity.

BIG O NOTATION

A nice trick when trying to determine the running time of a program is to look for loops. If the program is looping over the input, it is likely O(N).

Cool, so if a loop means that the relation is linear. What would this be classified as?

BIG O NOTATION

```
function sumCharCodes(needle) {
  let sum = 0;

  for (let i = 0; i < needle.length; i++) {
    sum += needle.charCodeAt(i);
  }

  for (let i = 0; i < needle.length; i++) {
    sum += needle.charCodeAt(i);
  }

  return sum;
}
```

Solvd
NodeJS LABA

BIG O NOTATION

You might be tempted to say O(2N), since we are looping over the input twice. And although theoretically you'd be correct, in Big O, another rule is to always drop constants.

O(2N) -> O(N)

And this makes sense, Big O is meant to describe the upper bound of the algorithm, the constant eventually becomes irrelevant as input grows.

BIG O NOTATION

Take the following examples:

N = 1; O(10N) = 10; O(N^2) = 1

N = 5; O(10N) = 50; O(N^2) = 25

N = 100; O(10N) = 1,000; O(N^2) = 10,000 (10x bigger)

N = 1,000; O(10N) = 10,000; O(N^2) = 1,000,000 (100x bigger)

N = 10,000; O(10N) = 100,000; O(N^2) = 100,000,000 (1000x bigger)

BIG O NOTATION

An important thing to notice is that there is practical vs theoretical differences. Just because O(N) is faster than O(N^2), doesn’t mean that it is practically faster for smaller input.

Remember, we drop constants, Therefore O(1000N) is faster than O(N^2) theoretically, but practically speaking, if N is small enough, the N^2 solution would be faster

BIG O NOTATION

Ok, now that we know all that, let’s do another example

```
function sumCharCodes(needle) {
  let sum = 0;

  for (let i = 0; i < needle.length; i++) {
    const charCode = needle.charCodeAt(i);

    // Capital S
    if (charCode === 83) {
      return sum;
    }

    sum += charCode;
  }

  return sum;
}
```

BIG O NOTATION

That one can be a bit more tricky. But the answer is also O(N). That is because in Big O, we often consider the worst case. That is, the worst case for our program is when a string doesn’t have an ’S’ in it.

Which means it will loop over the entire string. Hence O(N)

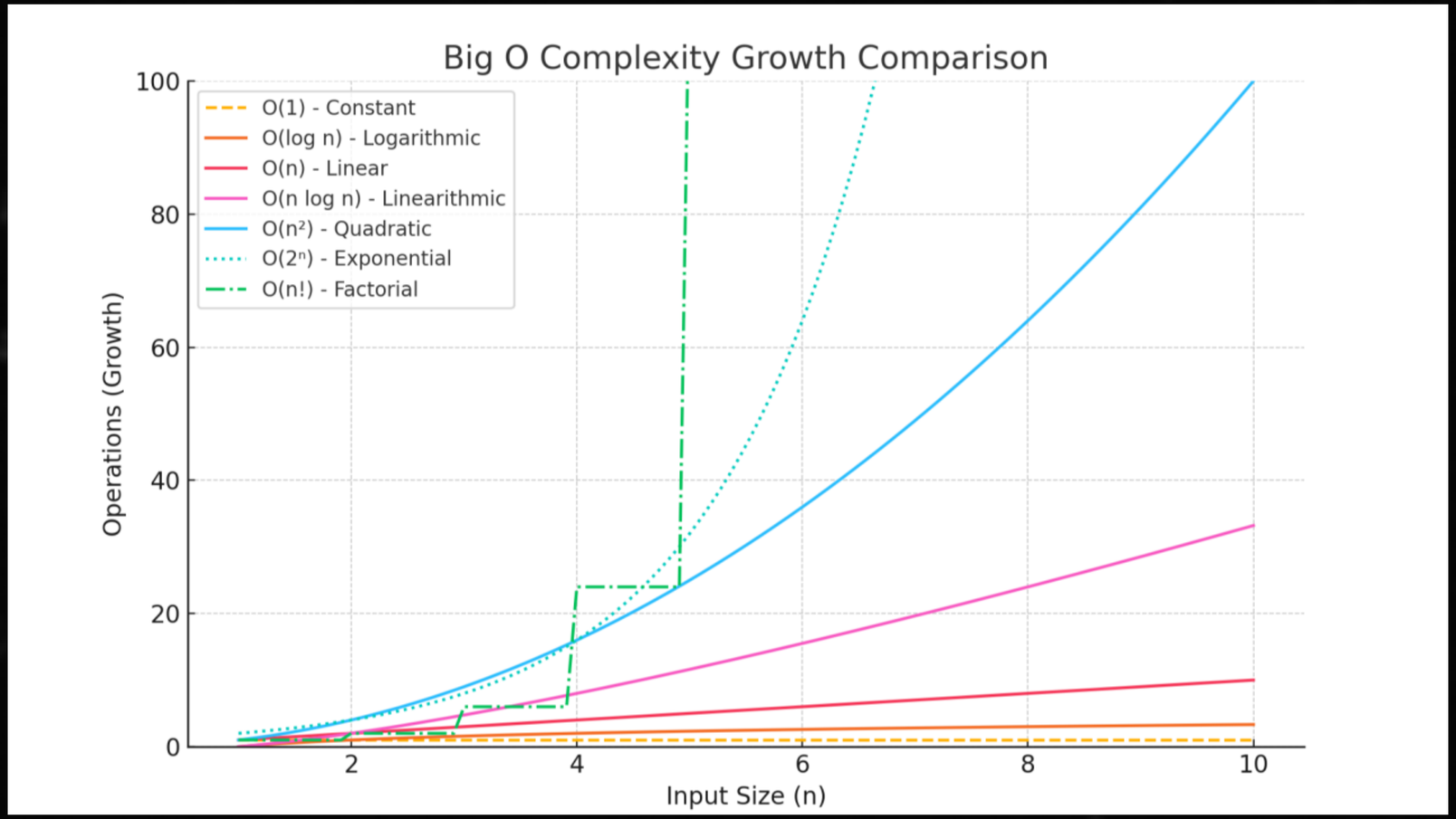
BIG O NOTATION

Key concepts to understand:

- Growth is with respect to input.
- Constants are dropped.
- Measure the worst case

(most of the time)

BIG O NOTATION



Solvd
NodeJS LABA

BIG O NOTATION

Great, lets see some more examples:

```
function sumCharCodes(needle) {
  let sum = 0;

  for (let i = 0; i < needle.length; i++) {
    for (let i = 0; i < needle.length; i++) {
      sum += needle.charCodeAt(j);
    }
  }

  return sum;
}
```

Solvd NodeJS LABA

```
function partition(arr, low, high) {
    let pivot = arr[high];
    let i = low - 1;

    for (let j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            [arr[i], arr[j]] = [arr[j], arr[i]];
        }
    }

    [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];
    return i + 1;
}

function quickSort(arr, low, high) {
    if (low >= high) return;
    let pi = partition(arr, low, high);

    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
}
```

BIG O NOTATION

Wait, is there anything else besides Big O? Well… Yes, there are countless ways to determine the running time of an algorithms. But it’s generally easier to consider the upper bound.

THE REAL TAKEAWAY

```
const Component = (props) => {
  return (
    <OtherComponent {...props} />
  )
}
```

Please don’t do this

Solvd NodeJS LABA