Big o notation allows us to talk about how the runtime of an algorithm grows as the input grows. We talk about the trend.

Time complexity:

We say that an algorithm is O(f(n)) if the number of simple operations the computer has to do is eventually less than a constant times f(n) as n increases.

* f(n) could be linear=> f(n)=n
* f(n) could be linear=> f(n)=n
* f(n) could be constant=> f(n)=1
* f(n) could be something entirely different

Constant dont matter => O(2n) is O(n), O(600) is O(1)

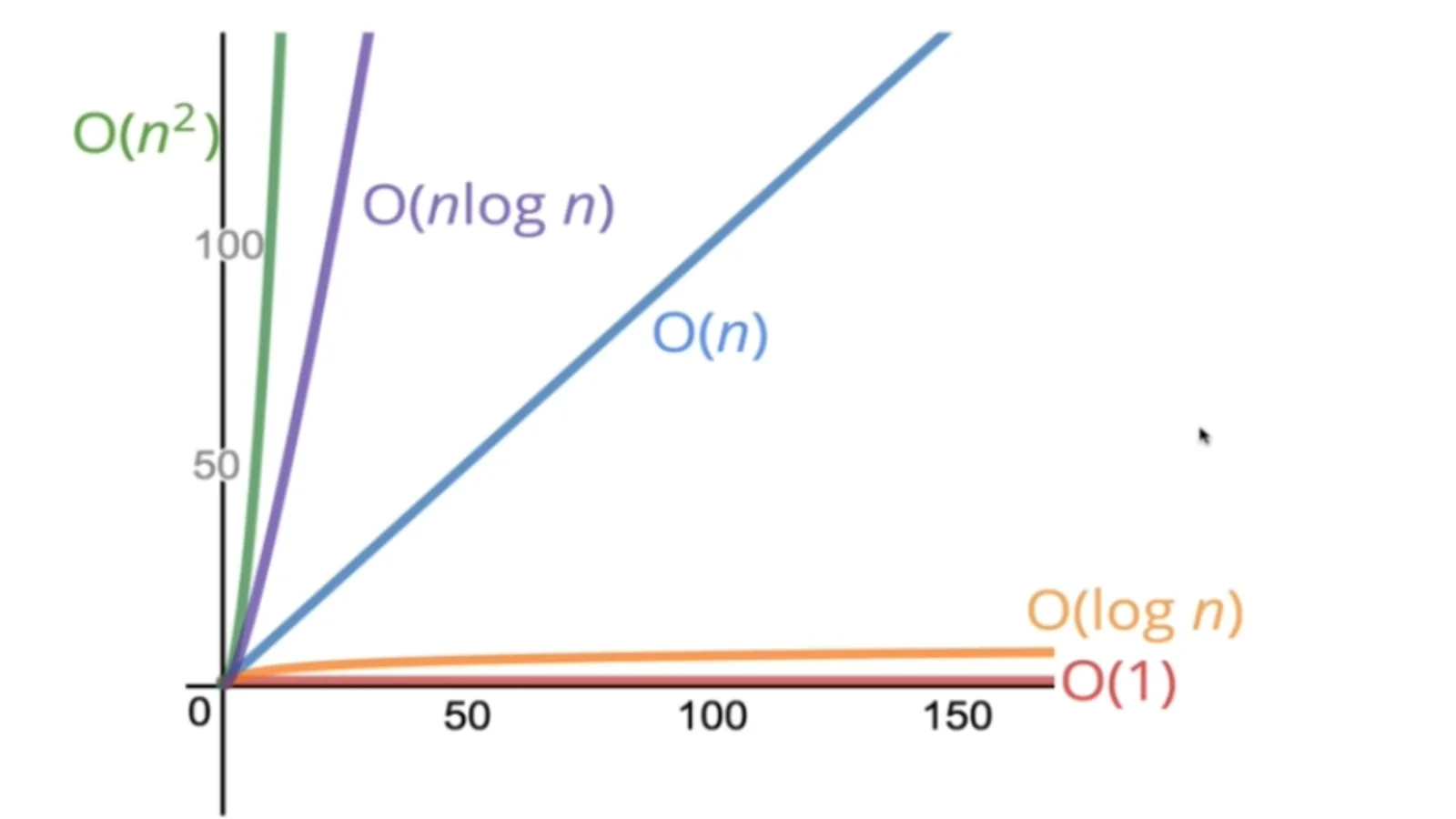
Smaller terms dont matter => O(n+10) is O(n), O(n2+5n+8) is O(n2)

Some shorthands:

1. Arithmetic operations are constant
2. Variable assignment is also constant
3. Accessing elements in an array (by index) or object (by keys) is constant
4. In a loop, the time complexity is the length of the loop times the complexity of whatever happens inside the loop.

Space complexity:

* Most primitives (booleans, numbers, undefined, null) are constant.
* Strings require O(n) space (where n is the string length).
* Reference types are generally O(n), where n is the length (for array) or the number of keys (for objects)



O(1) is the best and O(n2) is the worst for both time and space complexity

Objects:

Insertion => O(1) => because it just inserts the new value and is independent of the size of the object.

Removal => O(1) => because it just removes the value and is independent of the size of the object.

Access => O(1) => because it just finds value and is independent of the size of the object.

Searching => O(n) => it loops through the entire object and that’s why the bigger the object the more times it needs.

Object.keys => O(n)=> it prints out all the keys of the object, so the bigger the object more time it needs.

Object.values => O(n)=> it prints out all the values of the object, so the bigger the object more time it needs.

Object.entries => O(n)=> it prints out all the keys and values of the object, so the bigger the object more time it needs.

Object.hasOwnProperty => O(1)=> it just gets the desired key and prints out that particular property

Array:

Insertion and removal => it depends. If we want to insert and remove at the end then it’s O(1) because it will just remove or add an element at the end and this operation does not depend on the size of the array. But if we want to add or remove from the middle or at the beginning, it is O(n). because after removal or addition, JS has to do reindexing of the array elements.

Searching => O(n) => it loops through the entire array and that’s why the bigger the object the more times it needs.

indexing/accessing => O(1)=> because, For indexing JS doesn’t loop through all the values till it reaches that index, it just jumps to that index.

Rules of Big-O Notation

Let’s represent an algorithm’s complexity as f(*n*). *n* represents the number of inputs, f(*n*)time represents the time needed, and f(*n*)space represents the space (additional memory) needed for the algorithm. The goal of algorithm analysis is to understand the algorithm’s efficiency by calculating f(*n*). However, it can be challenging to calculate f(*n*). Big-O notation provides some fundamental rules that help developers compute for f(*n*).

*Coefficient rule:* If f(*n*) is O(g(*n*)), then kf(*n*) is O(g(*n*)), for any constant k > 0. The first rule is the *coefficient rule*, which eliminates coefficients not related to the input size, *n*. This is because as *n* approaches infinity, the other coefficient becomes negligible.

*Sum rule:* If f(*n*) is O(h(*n*)) and g(*n*) is O(p(*n*)), then f(*n*)+g(*n*) is O(h(*n*)+p(*n*)). The sum rule simply states that if a resultant time complexity is a sum of two different time complexities, the resultant Big-O notation is also the sum of two different Big-O notations.

*Product rule:* If f(*n*) is O(h(*n*)) and g(*n*) is O(p(*n*)), then f(*n*)g(*n*) is O(h(*n*)p(*n*)). Similarly, the product rule states that Big-O is multiplied when the time complexities are multiplied.

*Transitive rule:* If f(*n*) is O(g(*n*)) and g(*n*) is O(h(*n*)), then f(*n*) is O(h(*n*)). The transitive rule is a simple way to state that the same time complexity has the same Big-O.