



function logTime(arr) {

let numberOfLoops = 0

for (let i = 1; i < arr.length; i \*= 2) {

numberOfLoops++

}

return numberOfLoops

}

let loopsA = logTime([1]) // 0 loops

let loopsB = logTime([1, 2]) // 1 loop

let loopsC = logTime([1, 2, 3, 4]) // 2 loops

let loopsD = logTime([1, 2, 3, 4, 5, 6, 7, 8]) // 3 loops

let loopsE = logTime(Array(16)) // 4 loops

// TimeO(logN), space O(1)

[The Binary Search algorithm](https://www.doabledanny.com/binary-search-javascript) has a Big O (log(n)). If we input a sorted array of length 16 (i.e. the bottom level in the tree above), it would only take 4 steps (count to the top tree node) to find the number we were looking for.

Algorithms with logarithmic time are often “divide and conquer” style, meaning the data set is cut down/reduced upon each loop iteration. The algorithm has less data to deal with on each loop and so can find or sort things quickly.

function reverseArray(arr) {

let newArr = []

for (let i = arr.length - 1; i >= 0; i--) {

newArr.push(arr[i])

}

return newArr

}

const reversedArray1 = reverseArray([1, 2, 3]) // [3, 2, 1]

const reversedArray2 = reverseArray([1, 2, 3, 4, 5, 6]) // [6, 5, 4, 3, 2, 1]

Time O(n) and space complexity is O(1)

So far, we’ve only looked at custom functions, but it’s important to realise Big O also applies to built-in JavaScript functions, such as the array methods push, pop, unshift and shift.

In the code snippet below, we have a 4-item array, arr. If we push 5 onto the end of this array, then all we have to do is create a new place at the end of the array, give it an index, and put the value of 5 there. It doesn’t matter what the length of the array is, it will always be constant time - Big O(1). The number of operations is always the same – constant.

let arr = [1, 2, 3, 4]

// Adding and removing to the end of the array => Big (1) - constant time

arr.push(5) // [1, 2, 3, 4, 5]

arr.pop() // [1, 2, 3]

But say we want to add 0 to the front of the array with unshift(0). We would have to re-index every item in the array, as the first index (index 0) would now point to our newly added value (0). We’d have to add 1 to every index in the array as the old first item is now the second, the old second index is now the third and so on…

So unshifting and shifting have linear time complexities - Big O(n) - because the longer the input array, the more items have to be re-indexed.

// Adding and removing to front of array => Big O(n) - linear time

arr.unshift(0) // [0, 1, 2, 3, 4]

arr.shift() // [2, 3, 4]

function linearithmic(n) {

for (let i = 0; i < n; i++) {

for (let j = 1; j < n; j = j \* 2) {

console.log("Hello")

}

}

}

Time: O(n log(n)), Space: O(1)

The outer loop iterates through 0 to n linearly (n) and the inner loop is log(n) because j is getting doubled on each loop.

function multiplyAll(arr1, arr2) {

if (arr1.length !== arr2.length) return undefined

let total = 0

for (let i of arr1) {

for (let j of arr2) {

total += i \* j

}

}

return total

}

let result1 = multiplyAll([1, 2], [5, 6]) // 33

let result2 = multiplyAll([1, 2, 3, 4], [5, 3, 1, 8]) // 170

Time: O(n^2) Space: O(1)

function fibonacci(num) {

// Base cases

if (num === 0) return 0

else if (num === 1) return 1

// Recursive part

return fibonacci(num - 1) + fibonacci(num - 2)

}

fibonacci(1) // 1

fibonacci(2) // 1

fibonacci(3) // 2

fibonacci(4) // 3

fibonacci(5) // 5

Time: O(2^n), Space: O(1)

Exponential time complexity

function factorial(n) {

let num = n

if (n === 0) return 1

for (let i = 0; i < n; i++) {

num = n \* factorial(n - 1)

}

return num

}

Time: O(n!), Space: O(1)