Structy Course Notes

# Big O Basics

* notation used to describe the performance of algorithms
* emphasis on how performance scales with the input size
* approximation

## why use Big-O?

* allows us to compare performance of algorithms
* prefer algorithms that use less time/less memory
* doesn’t rely on the environment (language, hardware etc)

## Big-O simplification rules

* drop any constant factors O(4n) -> O(n), here 4 is a constant
* we’re trying to describe behaviour as input size grows so we can drop constants

// O(n/2) -> O(n)

* drop smaller terms in a sum of multiple terms:

// O(n\*\*2 + n) -> O(n\*\*2)

// O(n + n\*\*4 + n\*\*2) -> O(n\*\*4)

// O(n\*\*4 - n\*\*3) -> O(n\*\*4)

* a combined example:

// O(4n\*\*2 + n + 5) -> drop constants O(n\*\*2 + n + 5) then drop small terms O(n\*\*2)

* in an interview situation you would give simplified O notation so its easy to compare your algorithm

## Big-O classifications

// worse = more memory/time

// O(n!) -> 8! = 8\*7\*6\*5..\*1 = 40320 = factorial

// O(c\*\*n) -> O(2\*\*n), O(3\*\*n) = exponential

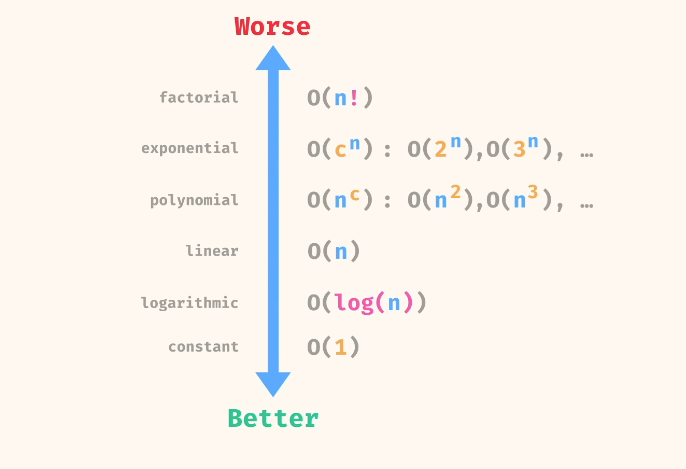
// O(n\*\*c) -> e.g. O(n\*\*2), O(n\*\*3) = polynomial

// O(n) = linear

// O(log(n)) -> 2\*\*5 =32 -> log\_base\_2(32)=5 (you have to divide 32 by 2 , 5 times to get to 1) = logarithmic (logarithmic is the opposite of exponential)

// O(1) = constant -> performance does not scale with the input size

// better = less memory/time



// O(2\*\*n + n\*\*10) -> O(2\*\*n) = here you drop the smaller term and keep the larger term = expontential is worse than polynomial so you keep exponential and drop polynomial

# Big-O Examples

* we are estimating how many operations an algorithm does relative to input size

## Constant time examples O(1)

* initializing a value:
* Mathematic operations (number or boolean operations):

Const a = 4

Const sum = a + 10

Console.log(sum)

* String indexing

Const str = ‘hello’

Console.log(str[1])

* Checking a key exists in an Object

Const stuff = {a:1, b:2, c:3}

Console.log(‘b’ in stuff)

## Linear time examples O(n)

* Checking an array for a value

Const colors = [‘red’, ‘blue’, ‘green’, ‘yellow’, ‘purple’]

Console.log(colors.includes(‘green’)

// it has to iterate through every element of the array

* For loops (single for loops)

Const array = [1,2,3,4,5,6,7]

Let sum = 0

For (let i=0; I < array.length; i++) {

Sum += array[i]

}

Console.log(sum)

* Using the split method

Const sentence = ‘hello world, how are you?’

Console.log(sentence.split(‘ ‘)

// split has to check every character for a space so it is O(n)

* The course will build a sense of time complexity of different built-in javascript methods

## Polynomial examples O(n\*\*c)

* Nested for loops

Const letters = ‘a’, ‘b’, ‘c’, d’, ‘e’, ‘f’]

For (let i=0; i< letters.length ; i++) {

For (let j=0; j< letters.length; j++ {

Console.log(letters[i], letters[j])

}}

// this has a time complexity of O(n\*\*2)

## Time and space complexity of a function

Const function1 = (array) => {

Return array.includes(‘potato’)

}

// length of input: n = array.length

// Time: O(n)

// Space: O(1)

* Another example

Const function2 = (n) => {

For (let I = 0; i< n/2; i++) {

Console.log(i)

}

// time: O(n/2) -> O(n)

// space: O(1)

* If you create arrays or strings that scale, then space complexity would be O(n)

Const function3 = (n) => {

Const nums = []

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

Nums.push(i)}

Return nums;

}

Console.log(function3(10))

// time: O(n) -> for loop is O(n) and push method is O(1)

// space: O(n) -> space complexity increases as n increases – it comes from the array that is created

Const function4 = (n) => {

Const nums = []

For (let i=0; i< n: i++) {

Nums.unshift(i);}

Return nums}

Console.log(function4(10))

// time: for loop = O(n), unshift method pushes new value to the front of array and reindexes every value in the array, so it is O(n) -> together they are O(n \* n) -> O(n\*\*2)