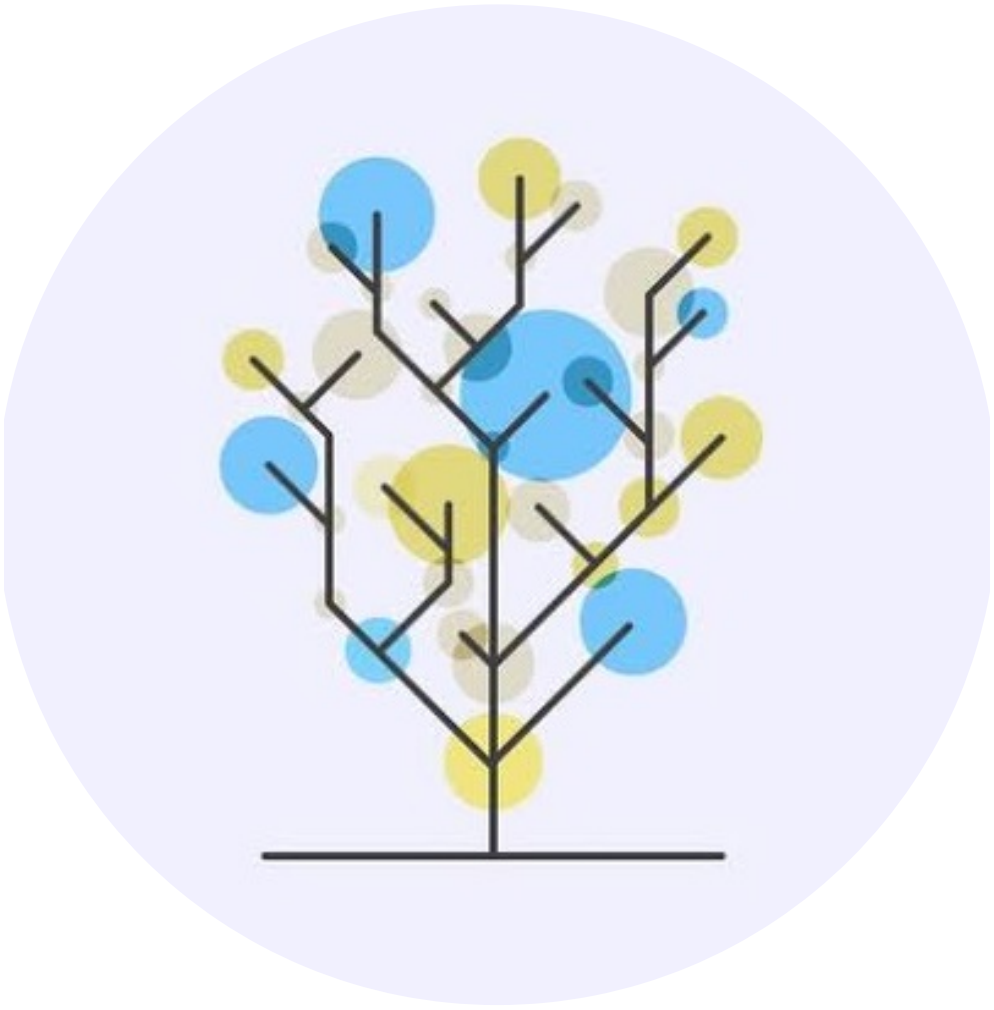

Big O Notation

02.01.2020

Data Structures & Algorithms

OVERVIEW

★Big O yazılan kodun verimliliği ile ilgili bir kavram

★ Mesela bir array içinde eleman arayan bir kodumuz olsun, bunun elmanı kaç saniyede bulduğu array boyutuna bağlıdır ayrıca CPU power vb. gibi bir çok dış faktöre de bağlıdır. Dolayısıyla kodun çalışma saniyesi ile verimlilik ölçmek anlamlı değil.

★ Bunun yerine **NUMBER OF OPERATIONS** ile verimlilik ölçeriz. Fonksiyon giren input'un eleman sayısı arttıkça, kodumuzun execute ettiği operasyon sayısının değişimi kodumuzu çok verimli veya çok verimsiz yapabilir.

★WHEN WE TALK ABOUT BIG O AND SCALABILITY OF CODE WE SIMPLY MEAN WHEN WE GROW BIGGER AND BIGGER OUR INPUT HOW MUCH DOES THE ALGORITHM SLOW DOWN. THE SLOWER IT SLOWS DOWN IS BETTER!

- **O(n):** Input ne kadar büyütse number of operations o kadar büyür. Loop ile tüm elemanları dönerek tek tek arama yapmak örnek verilebilir. Linear Time!
- **O(1):** Örneğin sürekli arrayin ilk elemanını yazdıran bir function olsun. Array size ne kadar olursa olsun, number of operation aynı kalır.

```
const performance = require('perf_hooks').performance;

const nemo = ['nemo'];
const everyone = ['dory', 'bruce', 'marlin', 'nemo', 'gill', 'bloat', 'nigel', 'squirt', 'darla',
'hank'];
const largeArray = new Array(100).fill('nemo');

function findNemo(array) {
  for (let i = 0; i < array.length; i++) {
    if (array[i] === 'nemo') {
      console.log('Found NEMO!');
    }
  }

}

findNemo(nemo); // This function has Big O notation of O(n) --> Linear Time

//Neden Linear çünkü, eğer array 3 elemanlı ise 3 operasyon gerekiyor, 4 ise 4, 5 ise 5, n ise n!

//n burada number of inputs olarak düşünülebilir.
```

```
// What is the Big O of the below function? Let's go line by line
function funChallenge(input) {
  let a = 10; // O(1)
  a = 50 + 3; // O(1)

  for (let i = 0; i < input.length; i++) { // O(n)
    anotherFunction();                     // O(n)
    let stranger = true;                   // O(n)
    a++;                                   // O(n)
  }
  return a;  // O(1)
}
// O(4n+3) ama günün sonunda göreceğiz ki bu aslında O(n)
//Ayrıca böyle tek tek hesaplamaya gerek kalmayacak onu da göreceğiz
```

```
function anotherFunChallenge(input) {
  let a = 5;        //O(1)
  let b = 10;       //O(1)
  let c = 50;       //O(1)
  for (let i = 0; i < input; i++) { //O(n)
    let x = i + 1; //O(n)
    let y = i + 2; //O(n)
    let z = i + 3; //O(n)

  }
  for (let j = 0; j < input; j++) { //O(n)
    let p = j * 2; //O(n)
    let q = j * 2; //O(n)
  }
  let whoAmI = "I don't know"; //O(1)
}

// O(7n+4)
```

★4 ve 5. derslerdeki örneklerde tek tek BigO hesapladık, bunu simplfy edebiliriz 4 rule var:

★ BIG O RULE 1: ALWAYS CONSIDER WORST CASE

- Burada olay şu: Mesela loop ile tek tek arama yapıyoruz, ama bizim elemanı bulunca break; diyoruz. Bizim eleman en baştaysa inputun boyutu kadar operasyon gerekmiyor o zaman O(n) yanlış mı?
- Hayır değil çünkü, BIG O hesaplamasında her zaman en kötü durum göz önüne alınır, yani arama algoritması için, aranan elemanın en sonda olduğunu düşünürüz!

★ BIG O RULE 2: REMOVE CONSTANTS

- Tek tek saydık ve Big O yu (n/2 + 7) falan bulduk diyelim burada önemli olan "n" çünkü biz karakteristiği merak ediyoruz exact number önemli değil! Bu yüzden n'in katsayısını ve sabit sayıları saymadan hesaplama yapabiliriz.

★ BIG O RULE 3: DIFFERENT TERMS FOR INPUTS

- Function içine iki farklı input alıyor diyelim array1 ve array2 o ve ikisi için de ayrı ayrı for looplar ile tüm elemanları ekrana yazdırsın bunun Big O su
- O(2n) > O(n) olmaz. O(a,b) veya O(n1,n2) olur!
- **What if loops are nested?**
  - Aynı hiearcy deki operasyonlar için toplama işlemi kullanırız ancak iç içe olanlar için çarpım kullanırız.
  - Mesela {'a','b','c','d','e'} şeklinde bir arrayimiz olsun bunu function'a input olarak alıyoruz ve elemanları yanyana yazdırmak istiyoruz yani aa ab ac ad ae bb ba bc ... şeklinde.
  - Burada O(n^2) oluyor! Verimsiz bir Big O!
  - Farklı inputlar alıp bunları iki loop içinde yazdırsaydı:
    O(n1*n2) olurdu!

★ BIG O RULE 4: DROP NON-DOMINANT TERMS

- Aslında bu Rule2 ile benzer. Basitçe olay, baskın term'i almak ve gerisini atmak.
- Mesela n^2 + 100n +100 var ise buna n^2 diyeceğiz!

```
//RULE 2 EXAMPLE
function printFirstItemThenFirstHalfThenSayHi100Times(items) {
    console.log(items[0]); // O(1)

    var middleIndex = Math.floor(items.length / 2);
    var index = 0;
    //Assignments ve small calculations sayılmadı bu ex. için.

    while (index < middleIndex) {   // O(n/2)
        console.log(items[index]);
        index++;
    }

    for (var i = 0; i < 100; i++) { // O(100)
        console.log('hi');
    }
}
//Yani O(n/2 + 101) geldi ama O(n) deriz önemli olan karakteristik!
```

```
//Example of O(n^2)
//Log all pairs of array

const boxes = ['a', 'b', 'c', 'd', 'e'];
function logAllPairsOfArray(array) {
  for (let i = 0; i < array.length; i++) {
    for (let j = 0; j < array.length; j++) {
      console.log(array[i], array[j])
    }
  }
}

logAllPairsOfArray(boxes)
//O(n^2) olacak!
```

★ Bu adreste BigO tablosu ve hangi datastructures hangi operation için nasıl bir BigO'ya sahip görünüyor:

★ **DATA STRUCTURES:** are simply ways to store data.
★ **ALGORITHMS:** ways to use data structures.

★**DATA STRUCTURES + ALGORITHMS = PROGRAMS**

★İkisini de iyi seçmeliyiz ve en iyi BigO ile programı tamamlamalıyız.

TIME & SPACE COMPLEXITY

★ Şimdiye kadar BigO ve Scalability kavramlarını açıklarken yalnızca zaman bağlamnında konuştuk yani kodumuzun hızlı olmasının iyi olacağını söyledik fakat başka bir boyut daha var bu da **MEMORY**.
★ Bilgisayardaki limitler - **zaman** (CPU kısıtlar) ve **hafıza** (RAM kısıtlar) - arasında trade-off olur!

★ 3 PILLARS OF PROGRAMMING

- Readable codes
- Efficient BigO Time Complexity
- Efficient BigO Space Complexity

★ Memory için BigO mantığı aslında aynı:
★Önceden input size ile problemi çözmek için kaç operasyon yapıldığına bakıyorduk.
★Program çalışırken bir şeyleri iki farklı şekilde hatırlar:

- **HEAP:** usually where we store variables that we assign values
- **STACK:** usually where we keep track of our function calls

★ REASONS FOR TIME COMPLEXITY

- Operations(+,-,*,/)
- Comparisons(<,>,==)
- Looping(for,while)
- Outside Function call (function())

★ REASONS FOR SPACE COMPLEXITY

- Adding Variables
- Adding Data Structures (Like arrays,objects,hashtables...)
- Function calls
- Allocations

DATA STRUCTURES OVERVIEW

★**Data Structure:** is collection of values!
★**Algorithms:** are the steps or processes we put into place to manipulate these collection of values
★**Data Structures + Algorithms = Programs**

★Bunların güzel yanı tüm dillerde aynı olması, bu bilgiler temel bilgiler.

★Data Structure is a collection of values. The values can have relationships among them and they can have functions applied to them.
★Each data structure is different in what it can do and what is it best used for.
★Each data structure is good and is specialized for its own.

★To understand better we can think of data structures as any sort of containers:

- Backpack: for schoolbooks
- Drawers: for clothes
- Fridge: for food
- Folder: for files
- Box: for toys

★Hepsinin kendine has işlevi var. Data structure'ı da aynen böyle düşünebiliriz.
★It is a way for us to organize data.
★ We can put things to data structures and we can take things from data structures.

★Tonlarca data structures var ama önemli olan 5-6 tane var biz onlardan bahsediyor olacağız.
★Blockchain de aslında simply a Data Structure. A way to hold information.

★Bu notlar kapsamında aşağıdaki data structures'dan bahsedilecek.

- Arrays
- Stacks
- Queues
- Linked Lists
- Trees
- Tries
- Graphs
- Hash Tables

★Bunun yanında algoritmalar olarak da aşağıdakilerden bahsedilecek.

- Sorting
- Dynamic Programming
- BFS + DFS (Searching)
- Recursion

★Each language has its own **built-in datatypes**.
★Javascript, for example, has **numbers**, **strings**, **booleans**, **undefined**
★ Each language has **data structures** to organize these **datatypes**. For example, in Javascript, we have **Arrays[ ]** and **Objects{ }**. Sanırım Arrays ve Objects de datatypes oluyor.

- numbers, strings, booleans, undefined primitive data types
- Arrays[ ] and Objects{ } complex data types

★ Most languages have enough data structures and data types for us to build our own data structures.

★What are the various operations that can be performed on different data structures?

★Data structures we will see are simply variations of how we stored the data on our computers.

★ SOME DATA STRUCTURES ARE GOOD AT SOME OPERATIONS OTHERS ARE GOOD AT OTHER OPERATIONS

- **INSERTION** is one type of action we are gonna perform with data structures. Adding new data item in a given collection of items.
- **DELETION**
- **TRAVERSAL:** Simply means access to each data item exactly once so that it can be processed.
- **SEARCHING:** We wanna find out the location of the data item if it exists in a given collection.
- **SORTING**
- **ACCESS:** Main one and probably the most important one.

More blogs

© Newtodesign.com All rights received.