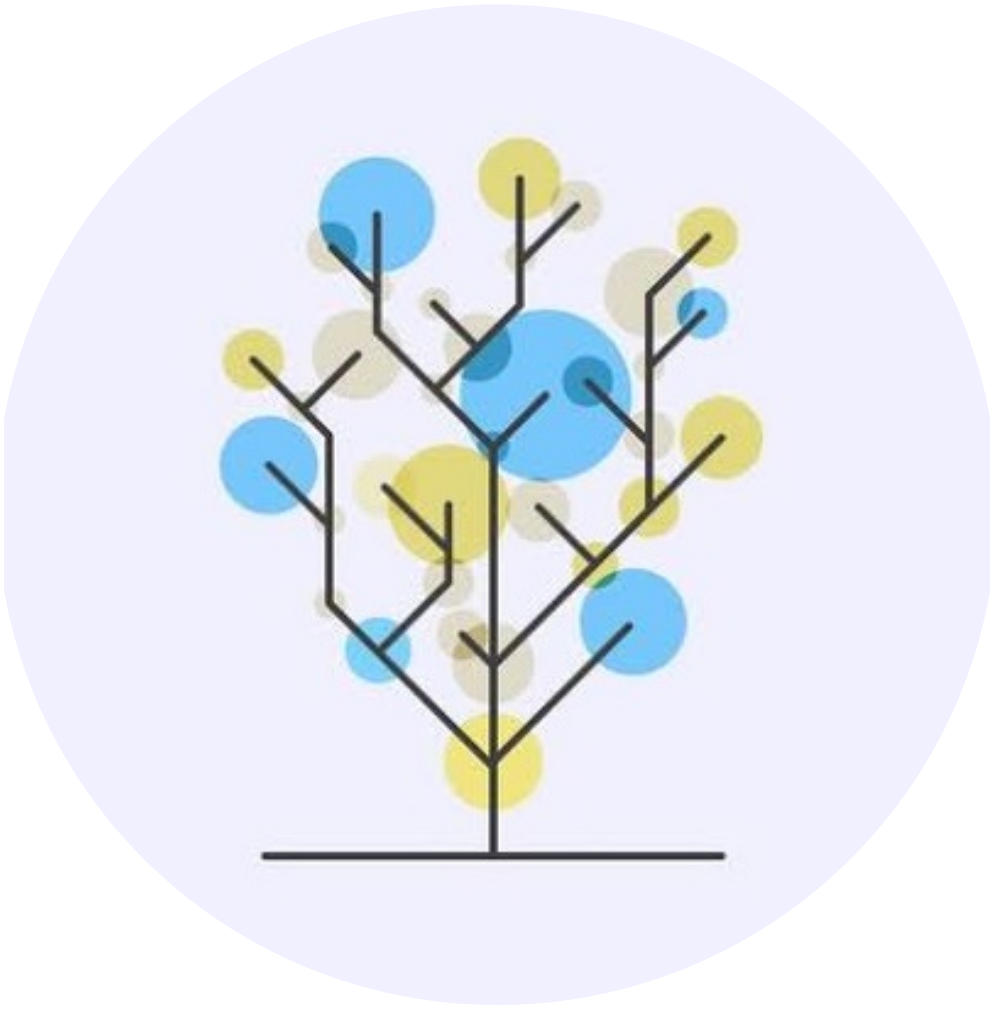


Recursion

05.01.2020

Data Structures & Algorithms

Algorithms Intro

- En çok kullanilan algoritmalarla (mülakatlarda) ilgilenecegiz:
  - Sorting
  - Dynamic Programming
  - Searching
  - Recursion
- Aslinda bunlari bilmeye de biliriz, bunlarin önemi Big O section'indan hatirlayacagimiz gibi, program büyüdükçe ortaya çikiyor. Yani büyük firmalarda önemli olur.

Recursion

- Recursion isn't exactly an algorithm, it's more of a concept that we are going to use throughout this algorithm section.
- Recursion is simply: a function that refers to itself inside of the function.
- Recursion is good for tasks that have repeated subtasks to do.
- The concept of recursion will be used in sorting and searching algorithms.

Stack Overflow

- Recursive bir function olsun, sürekli kendini çagirsin. Bu function bir kez çagrildiginda, sonsuz bir loop yaratir. Eskiden bu sebeple program çökebilirdi.
- Şuanda stack overflow hatasi döndürür.
- Bu şöyle düşünülebilir, bir function içinden yenisi çagirildiginda stack yapisina [illegible] bu fonksiyonlar kaydedilir ki yeni fonksiyon tanimlaninca stack'ten eski fonksiyon alinir ve kaldigi yerden devam edilir.
- Recursion olayinda bu stack hiç bosalmaz dolayisiyla limiti doldugunda stack overflow gerçekleşmiş olur.
- Stack overflow recursion'in en önemli problemlerinden biridir. Programa zarar verecek sonsuz looplar yaratabiliriz aynen bu recursion olayi costs memory.

Anatomy of Recursion (Javascript)

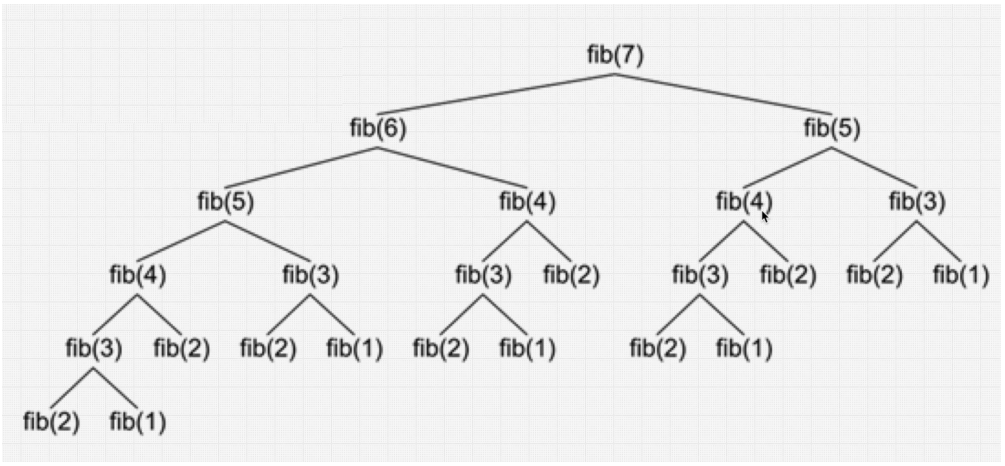
```
//Anatomy of Recursion

//Every recursive function need to have something called a BASE CASE or STOP POINT.

//Her iterasyon için recursive function'ın iki seçeneği var:
    // 1) Recursive case: Call the function again and run it.
    //  2) Base case: Stop calling the function.

// Here is an example of recursive function in JS:


let counter = 0;
function inception(){
    debugger; // for debugging in chrome console.
    if (counter>3) {
        return 'done!'
    }
    counter++;
    inception();
}

// Bu ifadeyi chrome console'a yapıştırıp fonksiyonu çağırabiliriz.
inception()

//Böylece her recursion için debugger satırından itibaren adım adım programı ilerletebiliriz.

//Böyle yapınca göreceğiz ki her counter 4 olana kadar sürekli yeni fonksiyon çağırılıyor ve bunlar stack'e kaydoluyor.

//counter 4 olunca if'e giriliyor en son fonksiyon 'done!' döndürüyor ve bir önceki fonksiyona geçiliyor ancak bir önceki fonksiyonda return ifadesi olmadığı için undefined döndürüyor ve geriye doğru undefined döndürülerek geliniyor.

//Ancak genelde, son fonksiyonun döndürdüğü değeri geriye doğru taşımak isteriz bu yüzden yukarıdaki kod bloğunda recursion'ın gerçekleştiği satırda bir değişiklik yapacağız ve return ifadesini ekleyeceğiz:

let counter = 0;
function inception(){
    debugger;
    if (counter>3) {
        return 'done!'
    }
    counter++;
    return inception();
}

inception()

//Böylece son fonksiyonun return'u bir önceki tarafından da return edilecek ve böylece en başa kadar taşınacaktır!

//Python'da return ifadesi eklenmese dahi son döndürülen 'done!' taşınıyor.
```

Same Recursion Code in Python

```
""" RECURSION """
counter = 0

def inception():
    global counter

    if counter>3:
        return 'done!'

    counter += 1;
    return inception()

inception()
```

Factorial Example (Python)

```
""" Factorial Example """

# Faktöriyel hesaplayan 2 fonksiyon yazalım, biri recursion ile diğeri iterative olarak bu işi yapsın:
# İkisi için de time complexity O(n) olur.

def findFactorialIterative(n):
    fact = 1
    for i in range(n,0,-1):
        fact *= i
    return fact

def findFactorialRecursive(n):

    if n == 1:
        return n
    else:
        return n*findFactorialRecursive(n-1)


# Diyelim ki n=3 olarak verildi ve fonksiyon çağırıldı:
# return 3*func(2) olur where func(2) --> 2*func(1) return eder --> func(1) 1 olarak döner ve geri dönüş başlar
# func(1)'den dönen 1, 2*func(1)'i 2 olarak döndürür ve bu da 3*func(2)'yi 6 olarak döndürerek bizi sonuca ulaştırır.
```



Fibonacci Example (Python)

```
""" Fibonacci Example """

#Given a number N return the index of the Fibonacci sequence, where the sequence is:

#  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...
#  The pattern of the sequence is that each value is the sum of the 2 previous values, that means that for N=5 -> 2+3

#   Function'a input olarak index verilecek karşılığında fibonacci serisinde o index'e karşılık düşen sayı bulunacak.


def fibIterative(index): """ O(n) TIME COMPLEXITY """

    if index == 0:
        return 0
    N = 1
    prev = 0
    for i in range(1,index):
        Nnew = N + prev
        prev = N
        N = Nnew

    return N


def fibRecursive(index):  """ O(2^n) TIME COMPLEXITY """

    if index<2 :
        return index

    else:
        return fibRecursive(index-1) + fibRecursive(index-2)

fibIterative(4)
fibRecursive(4)

 """ fibRecursive'in time complexity'si O(2^n) çünkü biz fib 7'yi hesaplamak istersek
yani index olarak 7 verirsek,  7 için 5 ve 6 hesaplanacak sonra 5 için 3 ve 4 ve 6 için 4 ve 5
şeklinde dallanarak 1'e kadar gelecek bunun fotoğrafını aşağıya eklerim. Yani her level'da ikiye ayrılacak
ve iki ayrı hesaplama yapılacak bu yüzden O(2^n) oluyor !!! """

 """ Peki o zaman neden Recursion kullanıyoruz? Bundan az sonra bahsedeceğiz, artı ve eksileri vardır.
 bu noktada şunu da belirtmeliyiz ki bu örnekteki O(2^n) time complexity aslında Dynamic Programming
 ve Memoization kullanılarak O(n)'e düşürülebilir. Bundan da daha sonra bahsedeceğiz. """
```

Recursive vs Iterative

→ Anything you do with recursion CAN be done iteratively (looping).

→ Peki neden recursion kullanırız?

- For some problems recursion is easier to write
- Thus recursion allows DRY (Don't Repeat Yourself) codes rather than WET (Write Everything Twice)
- But causes larger stack. Everytime we add a function to the call stack it adds extra piece of memory.

→ Sonuçta iterative approaches tend to be more efficient because they don't make these additional function calls that take up the stack space.

→ Ancak recursive solutions might be more readable.

- Using recursion when you are working with data structures that you are not really sure how deep they are, where you don't know how many loops to go through.
- We'll see that recursion is really useful for things such as tree datastructures and doing traversal

Tail Call Optimization: Allows recursions to be called without increasing the call stack. Başka sekillerde de recursion'ı modifiye edenek more memory efficient yapabiliriz.

When to Use Recursion?

→ Traversing veya searching through trees or graphs gibi complex konularda recursion ds very very useful. Hatta better than iterative approaches.

→ Benzer sekilde sorting için de recursion'in tercih edildigi durumlar mevcut.

(*) Everytime you're using a tree or converting something into a tree, consider recursion.

when?
- Problem divided into a number of subproblems that are smaller instances of the same problem
- Each instance of the subproblem is identical in nature
- The solutions of each subproblem can be combined to solve the problem at hand

Divide and conquer using Recursion!

→ Traversing over a tree'yi looping ile yapmak recursion'a göre çok daha zor ve karisik.

Review

→ Stack overflow'a dikkat etmeli.

→ More readable codes but sometimes recursion is less efficient than iteration or looping

→ Recursion ile yapilabilen her sey iteratively yapilabilir.

→ Recursion and space complexity are not friends

→ Sorting veya tree traversal gibi konularda recursion makes things simpler.

- Merge Sort
- Quick Sort
- Tree Traversal
- Graph Traversal

konularinda recursion kullanacagiz.

More blogs