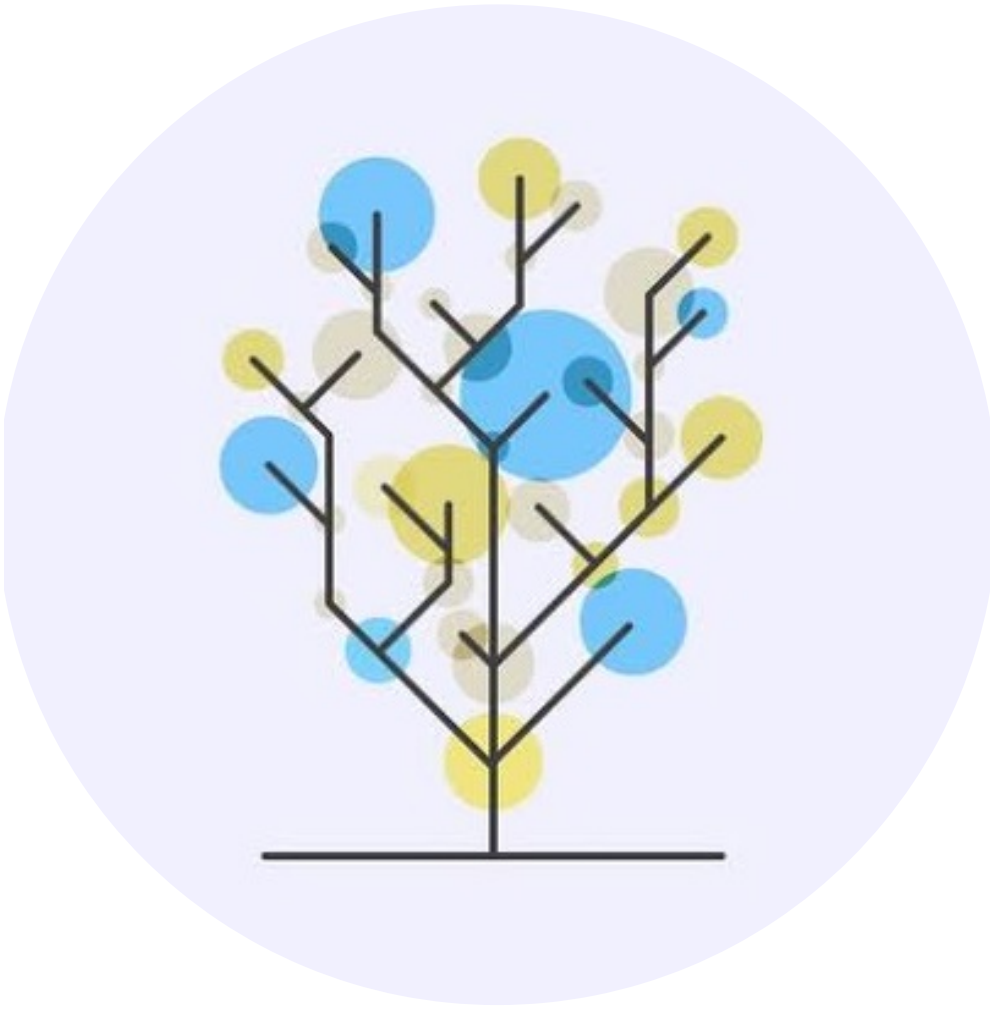


Dynamic Programming

06.01.2020

Data Structures & Algorithms

DYNAMIC PROGRAMMING

- Dynamic Programming -

- It is an optimization technique.
- İsmine çok takılmış, çok mantıklı değil.
- Aslında olay Recursion ile Cache ikilisine binanada kullanarak. Time Complexity'i optimize etmek.
- Dynamic Programming is a way to solve problems by breaking it down into a collection of subproblems, solving each of these subproblems just once and storing their solutions in case next time the same subproblem occurs.

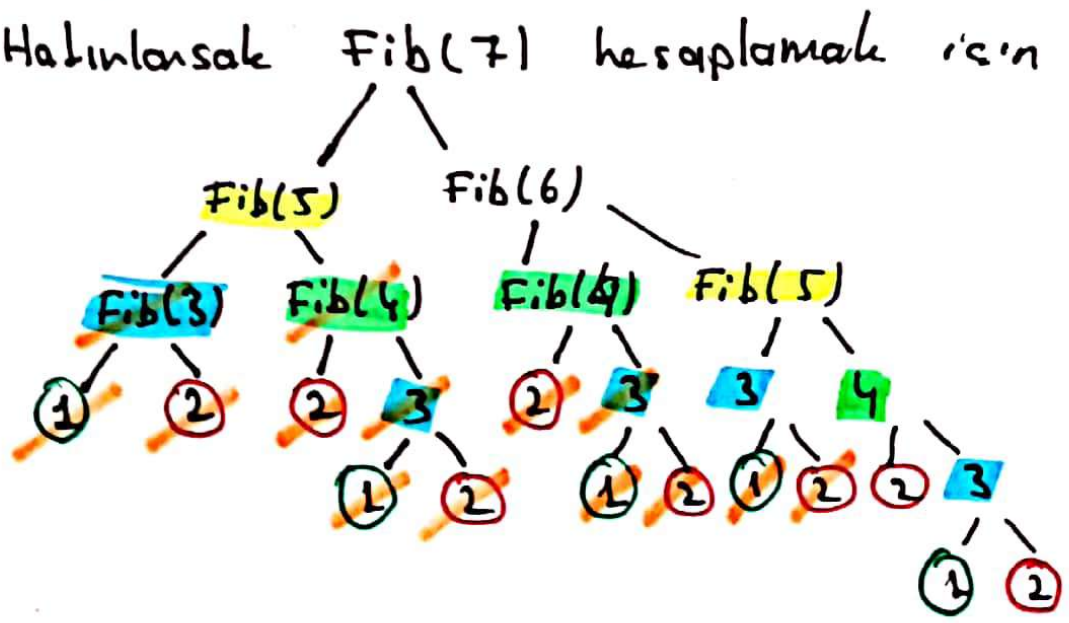
Memoization

Caching: is a way to store values to use them later on.

- It is a way for us to speed up programs.

- Memoization is a specific form of caching. We use it a lot in Dynamic Programming.

- Fibonacci örneğinde Time Complexity $O(2^n)$ idi bu O(n)'e kıyasla çok kötü çünkü gereken işlem sayısı çok hızlı artıyor.
  - Hatırlarsak Fib(7) hesaplamak için



- Memoizatin sayesinde her birini sadece bir kez hesaplamamız yetecekti. Bu da bize $O(2^n)$'den $O(n)$'e geçişi sağlar!

MEMOIZATION (Js)

```
//Let's say I have a function that adds 80 to the input as below:

function addTo80(n){
  console.log('Long time operation') //Say n+80 is an operation that takes a lot of time.
  return n+80;
}

//Diyelim ki bu fonksiyonu n=5 için farklı yerlerde çağırıyoruz:

console.log(addTo80(5))
console.log(addTo80(5))
console.log(addTo80(5))

// Her seferinde 'Long time operation' da gerçeklenmek zorunda kalır
// Daha önce yaptığımız işlemin aynısını tekrar tekrar yapmamız gerekir.

// ************************************************************************************

//BY USING MEMOIZATION WE CAN IMPROVE THIS SITUATION

let cache = {};
function memoizedAddto80(n){

  if (n in cache){ // cache objesinde (hashtable) n adında bir property halihazırda var mı?
    return cache[n]; //or cache[n]  cache objesinin n property'sinin değerini alıyoruz.
  } else{
    console.log('Long time operation') //Say n+80 is an operation that takes a lot of time.
    cache[n] = n+80; // ilgili n cache'de yoksa, hashtable'a ekleriz
    return cache[n];
  }
}

console.log(memoizedAddto80(5)) // Sadece bu işlem takes long time
console.log(memoizedAddto80(5)) // Sonuç hesaplanmaz cache'den alınır.
console.log(memoizedAddto80(5)) // Sonuç hesaplanmaz cache'den alınır.

// Memoization: is caching the return value of a function based on its parameters.
// If next time the function is called with same parameter, the result will be found in the cache,
and this will save us doing the same operations again and again.


// ************************************************************************************

// Yukarıdaki memoization örneğinde cache hashtable'ı fonksiyonun dışında, global scope'ta
tanımlandı.

// Ideally, it is good practice for cache to live inside the function without polluting the global
scope.

// Bu formu sağlamanın her dilde bir çok yolu var, js için ilk akla gelen yol:

function newMemoizedAddto80(n){
   let cache = {}; // Direkt buraya koyarsam olmaz çünkü fınksiyon her çağrıldığında cache
resetlenir, o yüzden cache'i hiç kullanamayız!!!
  if (n in cache){
    return cache[n];
  } else{
    console.log('Long time operation')
```

```
    cache[n] = n+80;
    return cache[n];
  }
}

// Closure denen yapıdan yararlanabiliriz.
function lastMemoizedAddto80(n){
  let cache = {};
  return function (n) { // function içinde yeni bir function çağrılıyor.
    if (n in cache){
      return cache[n];
    } else{
      console.log('Long time operation')
      cache[n] = n+80;
      return cache[n];
    }
  }
}

const memoized = lastMemoizedAddto80();

console.log(memoized(5)) // Sadece bu işlem takes long time
console.log(memoized(5)) // Sonuç hesaplanmaz cache'den alınır.
console.log(memoized(5)) // Sonuç hesaplanmaz cache'den alınır.
```

MEMOIZATION - Factorial Example (Python)

```python
# Create cache for known results
factorial_memo = {}

def factorial(k):

    if k < 2:
        return 1

    if not k in factorial_memo:
        factorial_memo[k] = k * factorial(k-1)

    return factorial_memo[k]

factorial(4)

##############################################################

#We can also encapsulate the memoization process into a class:
class Memoize:
    def __init__(self, f):
        self.f = f
        self.memo = {}
    def __call__(self, *args):
        if not args in self.memo:
            self.memo[args] = self.f(*args)
        return self.memo[args]

#Then all we would have to do is:
def factorial(k):

    if k < 2:
        return 1

    return k * factorial(k - 1)

factorial = Memoize(factorial)
```

FIBONACCI EXAMPLE (Dynamic Programming)

```
//0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...
// Without Memoization - Only recursive
let calculations = 0;
function fibonacci(n) { //O(2^n)

  if (n < 2) {
    return n
  }
  return fibonacci(n-1) + fibonacci(n-2);
}

// With memoization - Dynamic Programming
function fibonacciMaster() { //O(n)
  let cache = {};
  return function fib(n) {
    calculations++; // just to test how many calculations is needed while input is getting larger.
    if (n in cache) {
      return cache[n];
    } else {
      if (n < 2) {
        return n;
      } else {
        cache[n] = fib(n-1) + fib(n-2);
        return cache[n];
      }
    }
  }
}

// Just another way of solving the same problem
function fibonacciMaster2(n) {
  let answer = [0,1];
  for ( let i = 2; i <= n; i++) {
    answer.push(answer[i-2]+ answer[i-1]);
  }
  return answer.pop();
}

const fasterFib = fibonacciMaster();

console.log('Slow', fibonacci(35))
console.log('DP', fasterFib(100));
console.log('DP2', fibonacciMaster2(100));
console.log('we did ' + calculations + ' calculations');
```

Dynamic Programming

→ We can think of Dynamic Programming as combining

Divide & Conquer + Momoization

→ When can we use Dynamic Programming:

- Can the problem be divided into subproblems. Just like tree-like structures, smaller and smaller problems.
- Whic indicates Recursive Solutions
- Are there repeletive subproblems? Öyleyse we can memorize the subproblems.

More blogs