19/03/2023

Time and Space Complexity of Recursive Solutions.

Iterative

```
for (int i = 0 ; i < n ; i++) {
    - - -
}
```

TC = O(n)

We have already seen how can we find the time complexity of iterative code. Here we will discuss about the recursive code.

↳ function calling itself

```
main () {
    fun (n) ;
}
```

```
fun () {
    n == 0 → return ;
    fun (n-1) ;
}
```

↱ if condition check

fun (0) → k1 processing

fun (n-1) → k processing + m bytes allocation
fun (n) → k processing + m allocation
main()

Function call stack

LIFO → Last In First Out

All the entries will be gone from stack & this is known as stack unwinding.
There are various instances of function fun as they have different input parameters.

Ex→
```
void print (int a[], int n) {
        if (n == 0)
                return ;
        cout << *a ;
        print ( a+1, n-1) ;
}
```
k time

The above code is recursive code for linear traversal of an array.

$F(n) = k + F(n-1)$

↳ Why not a? The reason is as TC is function of n.

$F(n) = k + F(n-1)$ → (Recurrence relation)

We can also write it as

$T(n) = T(n-1) + k$

$T(n-1) = T(n-2) + k$

$T(n-2) = T(n-3) + k$

⋮

$T(1) = k + T(0)$

$T(0) = k1$

Add all above equations, and then find Time complexity.

After adding we get

$T(n) = nk + k1 \rightarrow O(n)$

This method is based on calculating through the formulae.

→ Visual method

* By recursive tree method

```
print(n)          k time
  ↓
print (n-1)       k time
  ↓
print (n-2)       k time
  ↓
  :
print (1)         k time
  ↓
print (0)         k1 time
```

k+k+--k
nk

TC = nk + k1 ⇒ O(n) Ans

Now let's find space complexity as operating system is maintaining a function call stack.

n space	1 -	print(0) → m byte
	1 -	print(1) → m byte
	2 -	⋮
	3	⋮
	n-2	print(n-2) → m byte
	n-1	print(n-1) → m bytes
	n	print(n) → m bytes
		main()

Space complexity = n + 1 → O(n)

↳ ignore in upper bound.

Ex→
```
int fact (int n) {
    if (n == 1)          } - k time
        return;          }
    return n * fact(n-1);
}
```

$T(n) = k + T(n-1)$
$T(n-1) = k + T(n-2)$
$T(n-2) = k + T(n-3)$
⋮
$T(1) = k$

Adding all the above equations

$T(n) = k + k + \_\_ k \rightarrow nk \rightarrow O(n)$

n
- fact(n) - k time
- ↓
- fact(n-1) - k time
- ↓
- fact(n-2) - k time
- ↓
- fact(n-3) - k time
- ⋮
- fact(1) - k time

$TC = n \times k \rightarrow O(n)$

Recursive tree method is much simpler as we can see from the above example.

fact(1)	→ m bytes
;	
fact(n-2)	→ m bytes
fact(n-1)	→ m bytes
fact(n)	→ m bytes
main()	

} n entries

Space complexity = $O(n \times m) = O(n)$

↳ constant

Ex → Recursive Binary Search

$$F(n) = k + F\left(\frac{n}{2}\right)$$

↳ Array size becomes half in binary search

$$T(n) = k + T\left(\frac{n}{2}\right)$$

$$T\left(\frac{n}{2}\right) = k + T\left(\frac{n}{4}\right)$$

$$T\left(\frac{n}{4}\right) = k + T\left(\frac{n}{8}\right)$$

⋮

$$T(2) = k + T(1)$$

$$T(1) = k$$

Add all the above equations

$$T(n) = x * k$$

$$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \cdots \rightarrow (1)\ \frac{n}{2^x}$$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

$$x = \log_2 n \quad \text{3 levels}$$

$$T(n) = (\log_2 n) \times k \rightarrow O(\log n)$$

↳ constant

BS(1)	→ m bytes
⋮	
BS(n/4)	→ m bytes
BS(n/2)	→ m bytes
BS(n)	→ m bytes
main()	

} $\log_2 n$

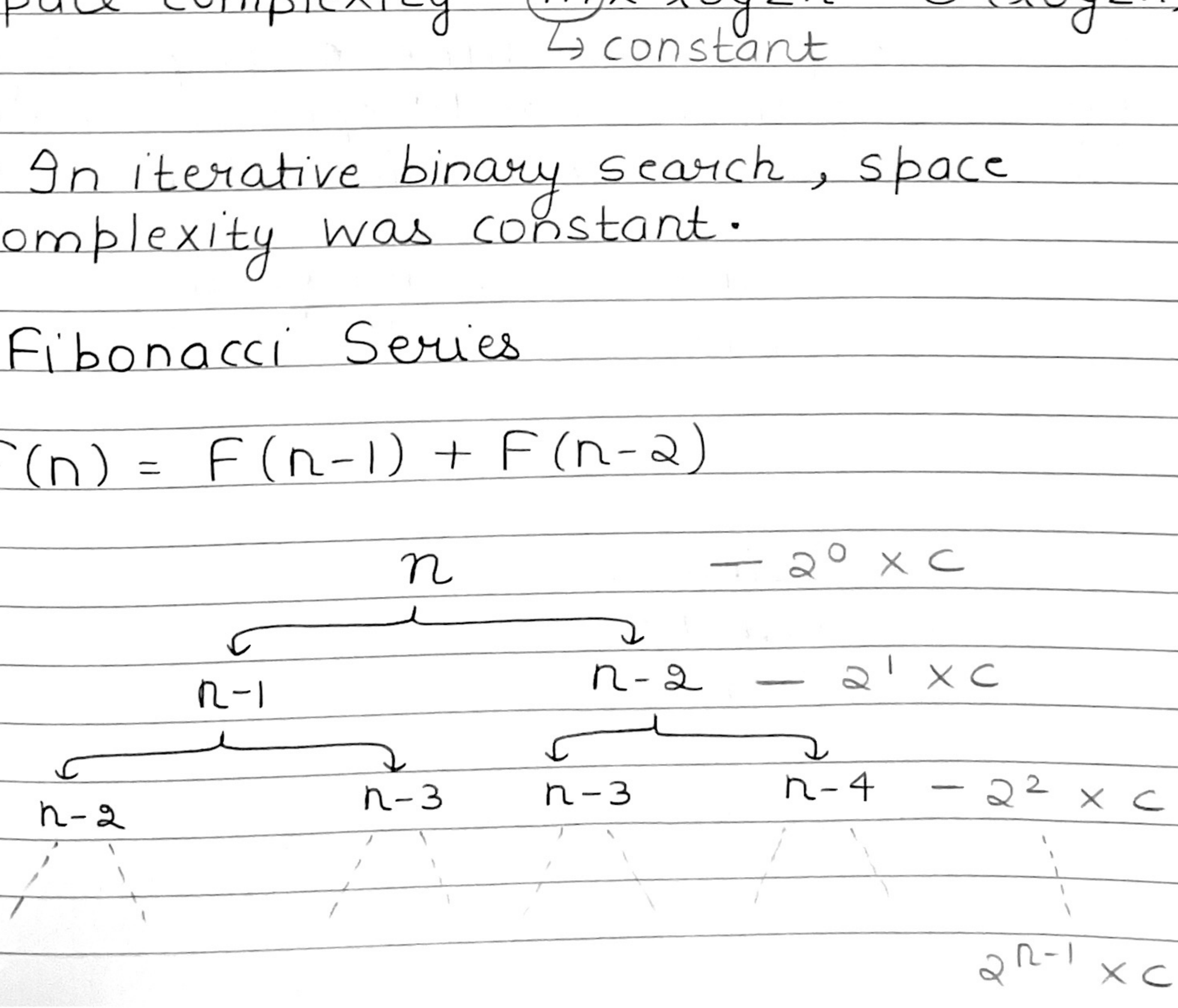
$$\text{Space complexity} = m \times \log_2 n = O(\log_2 n)$$

↳ constant

Note → In iterative binary search, space complexity was constant.

Ex → Fibonacci Series

$$F(n) = F(n-1) + F(n-2)$$



0 to n-1 ⇒ n levels

$T(n) \leqslant 2^0 c + 2^1 c + \cdots + 2^{n-1} c$

↳ because at last level all nodes may not be present.

$T(n) \leq c [2^0 + 2^1 + \_\_ + 2^{n-1}]$

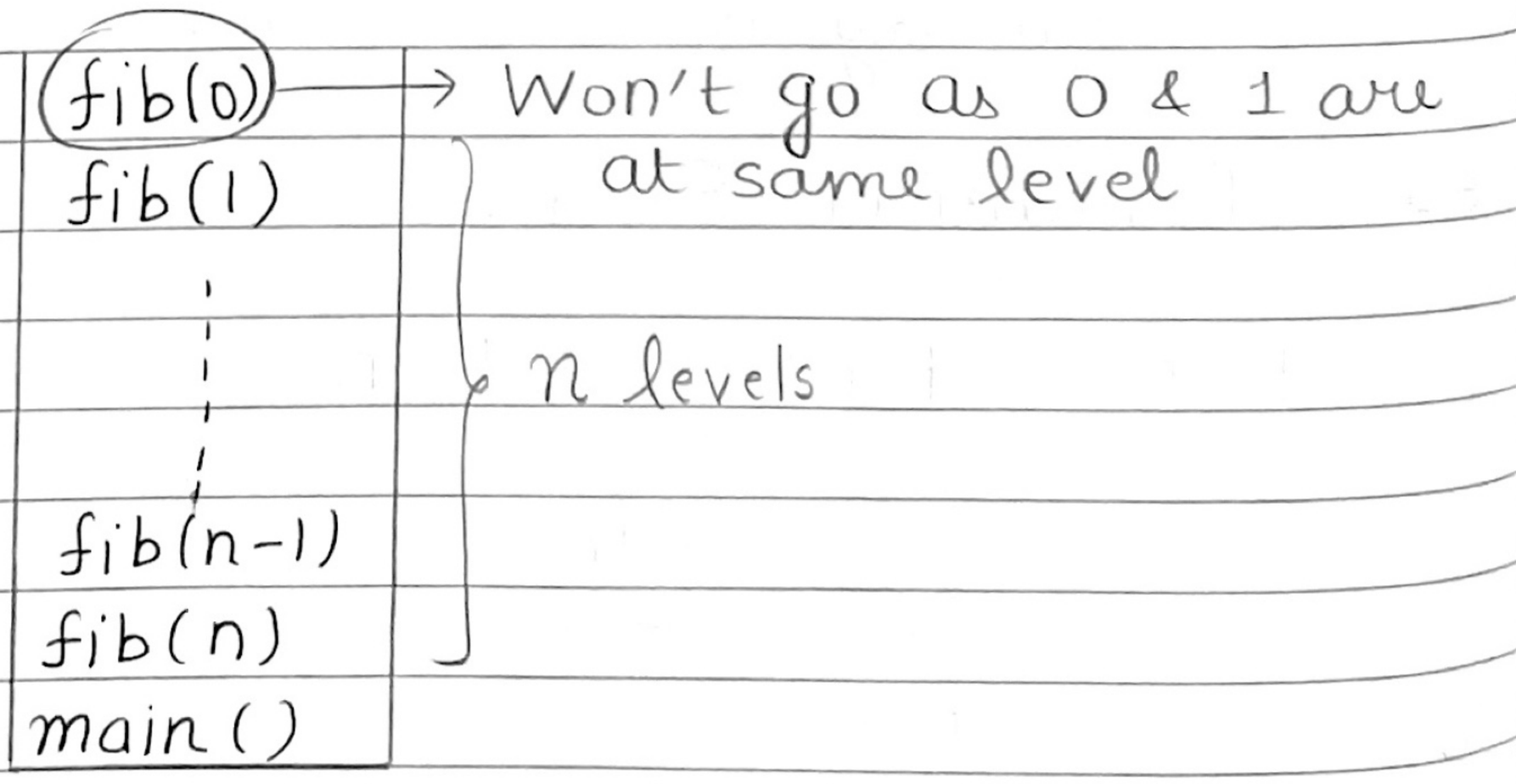
$\text{Sum of GP} = \frac{a(r^n - 1)}{r - 1}$

$r = 2 \quad , a = 1$

$T(n) \leq c \left[1 \times \left(\frac{2^n - 1}{2 - 1} \right) \right]$

$T(n) \leq c (2^n - 1) \rightarrow O(2^n)$

This is exponential time complexity. We will getting towards optimal solution in O(n) time with the help of DP.



Space complexity = O(n)