Lec 20: Dynamic Programming

Date: 18/01/23

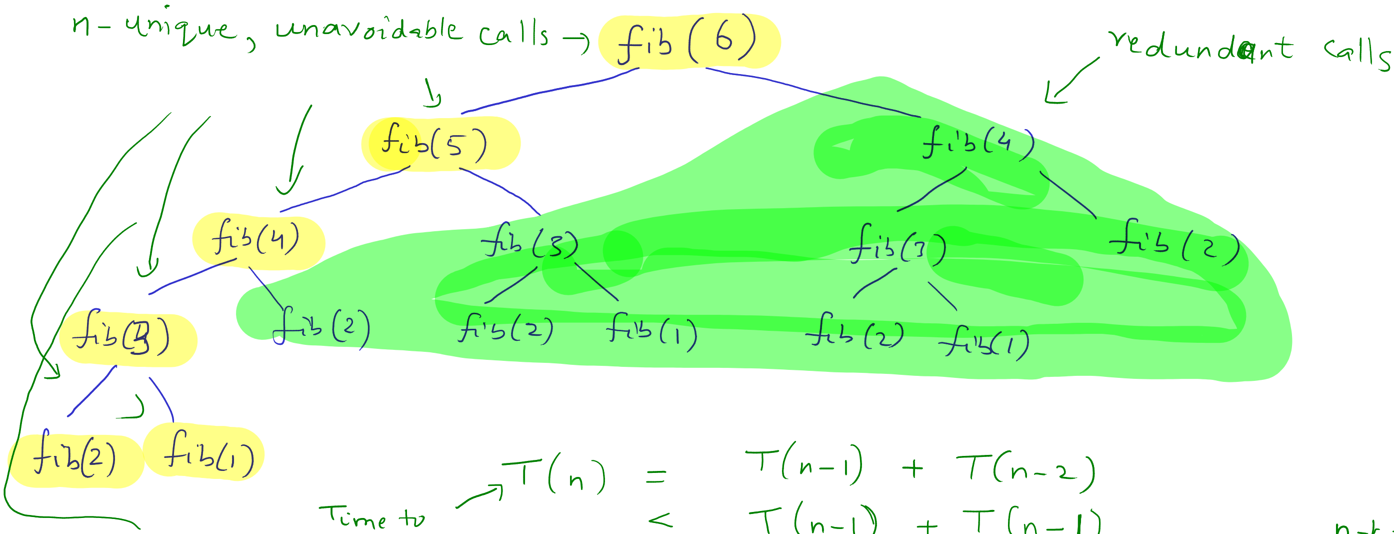
Fibonacci Sequence

1 1 2 3 5 8 13 21 . . . .

fib(1) fib(2) fib(7)

```
fib(n)
{
  if n==1 or n==2
     res=1
  else
     res=fib(n-1)
        +fib(n-2)
  return res
}
```

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \text{if } n > 2 \\ 1 & \text{if } \underbrace{n==1 \text{ or } n==2}_{n \leq 2} \end{cases}$$



Time to compute n-th Fibonacci number $\rightarrow T(n)$

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &< T(n-1) + T(n-1) \\ &= 2T(n-1) \\ &< 2(2T(n-2)) \\ &< 2^k T(n-k) \\ &= O(2^{n-2}) = O(2^n) \end{aligned}$$

Exponential time

$n-k == 1$, $n-k == 2$ } return 1

$T(n-k) = c$ when $n-k \leq 2$

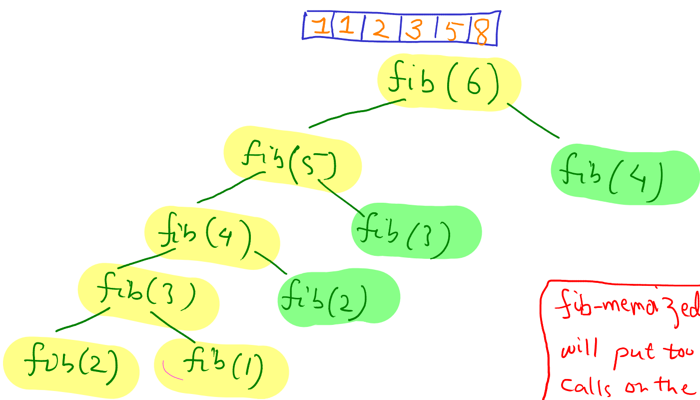
$\Rightarrow k \geq n-2$

Memoization

No redundant calls. (when available)

Save result of every fib(n) in memo(n) and reuse.

```
fib_memoized(n, memo)
    if memo[n] != null              <- O(1)
         return memo[n]             <- O(1)
    if n==1 or n==2                 <- O(1)
         res=1                      <- O(1)
    else
         res = fib_memoized(n-1)
               + (n-2)
    memo[n] = res                   <- O(1)
    return res                      <- O(1)
```



fib-memoized(1000) will put too many calls on the "recursion stack" and give an error.

$T(n) = O(n) << O(2^n)$ of non-memoized version.

- These recursive calls fill up the memo array.
- Memo has n entries
- So, such recursive calls cannot be more than n.

Bottom-up Approach: Fill memo in a bottom-up fashion.

```
fib_bottomup (n)
O(1)   if n==1 or n==2
           return 1
       memo = empty array of size n
       memo[1] = 1
       memo[2] = 1
O(n)   for i = 3 to n
           memo[i] = memo[i-1] + memo[i-2]
O(1)   return memo[n]
```

fib(6)

1	1	2	3	5	8

$O(n)$ in space

$O(n)$ in time

No recursion stack.
No calls.