

ALGORITMOS AVANÇADOS

Prof. Michael da Costa Móra

Programação Dinâmica

Programação dinâmica

"Dynamic programming is a fancy name for divide-and-conquer with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table. The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table. Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often. In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed."

I. Parberry, *Problems on Algorithms*, Prentice Hall, 1995.

Números de Fibonacci

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

Números de Fibonacci

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$$

n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

Algoritmo recursivo para F_n :

FIBO-REC (n)

```
1  se  $n \leq 1$ 
2      então devolva  $n$ 
3  senão  $a \leftarrow$  FIBO-REC ( $n - 1$ )
4           $b \leftarrow$  FIBO-REC ( $n - 2$ )
5      devolva  $a + b$ 
```

Consumo de tempo

FIBO-REC (n)

1 **se** $n \leq 1$

2 **então devolva** n

3 **senão** $a \leftarrow$ FIBO-REC ($n - 1$)

4 $b \leftarrow$ FIBO-REC ($n - 2$)

5 **devolva** $a + b$

n	16	32	40	41	42	43	44	45	47
tempo	0.002	0.06	2.91	4.71	7.62	12.37	19.94	32.37	84.50

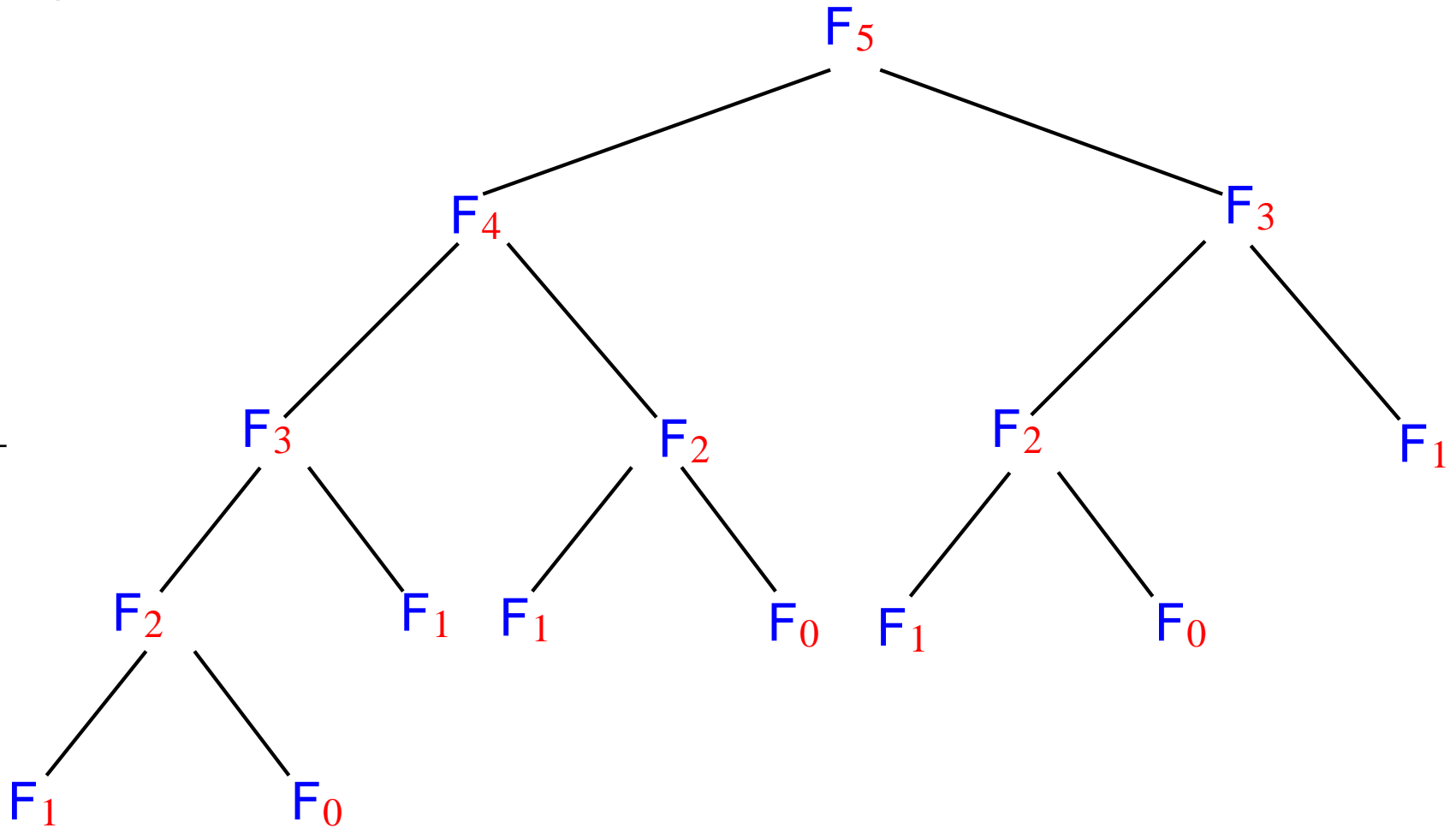
tempo em segundos.

$F_{47} = 2971215073$

Consumo de tempo

Consumo de tempo é **exponencial**.

Algoritmo resolve subproblemas muitas vezes.



Resolve subproblemas muitas vezes

```
FIBO-REC (5)
  FIBO-REC (4)
    FIBO-REC (3)
      FIBO-REC (2)
        FIBO-REC (1)
        FIBO-REC (0)
      FIBO-REC (1)
    FIBO-REC (2)
      FIBO-REC (1)
      FIBO-REC (0)
  FIBO-REC (3)
    FIBO-REC (2)
      FIBO-REC (1)
      FIBO-REC (0)
    FIBO-REC (1)
```

FIBO-REC(5) = 5

Resolve subproblemas muitas vezes

FIBO-REC (8)

FIBO-REC (7)

FIBO-REC (6)

FIBO-REC (5)

FIBO-REC (4)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (4)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (5)

FIBO-REC (4)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (6)

FIBO-REC (5)

FIBO-REC (4)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (4)

FIBO-REC (3)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

FIBO-REC (1)

FIBO-REC (2)

FIBO-REC (1)

FIBO-REC (0)

Memoização (sem o R mesmo!)

In computing, *memoization* or *memoisation* is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Although related to caching, *memoization* refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement. In the context of some logic programming languages, *memoization* is also known as *tabling*.

Algoritmo de programação dinâmica

FIBO (n)

1 $f[0] \leftarrow 0$

2 $f[1] \leftarrow 1$

3 **para** $i \leftarrow 2$ **até** n **faça**

4 $f[i] \leftarrow f[i - 1] + f[i - 2]$

5 **devolva** $f[n]$

Note a tabela $f[0 \dots n-1]$.

f					★	★	??			
-----	--	--	--	--	---	---	----	--	--	--

Consumo de tempo é $\Theta(n)$.

Algoritmo de programação dinâmica

Versão com economia de espaço.

FIBO (**n**)

1 **se** **n** = 0 **então devolva** 0

2 **f_ant** \leftarrow 0

3 **f_atual** \leftarrow 1

4 **para** **i** \leftarrow 2 **até** **n** **faça**

4 **f_prox** \leftarrow **f_atual** + **f_ant**

5 **f_ant** \leftarrow **f_atual**

6 **f_atual** \leftarrow **f_prox**

7 **devolva** **f_atual**

Versão recursiva eficiente

MEMOIZED-FIBO (f, n)

```
1  para  $i \leftarrow 0$  até  $n$  faça
2       $f[i] \leftarrow -1$ 
3  devolva LOOKUP-FIBO ( $f, n$ )
```

LOOKUP-FIBO (f, n)

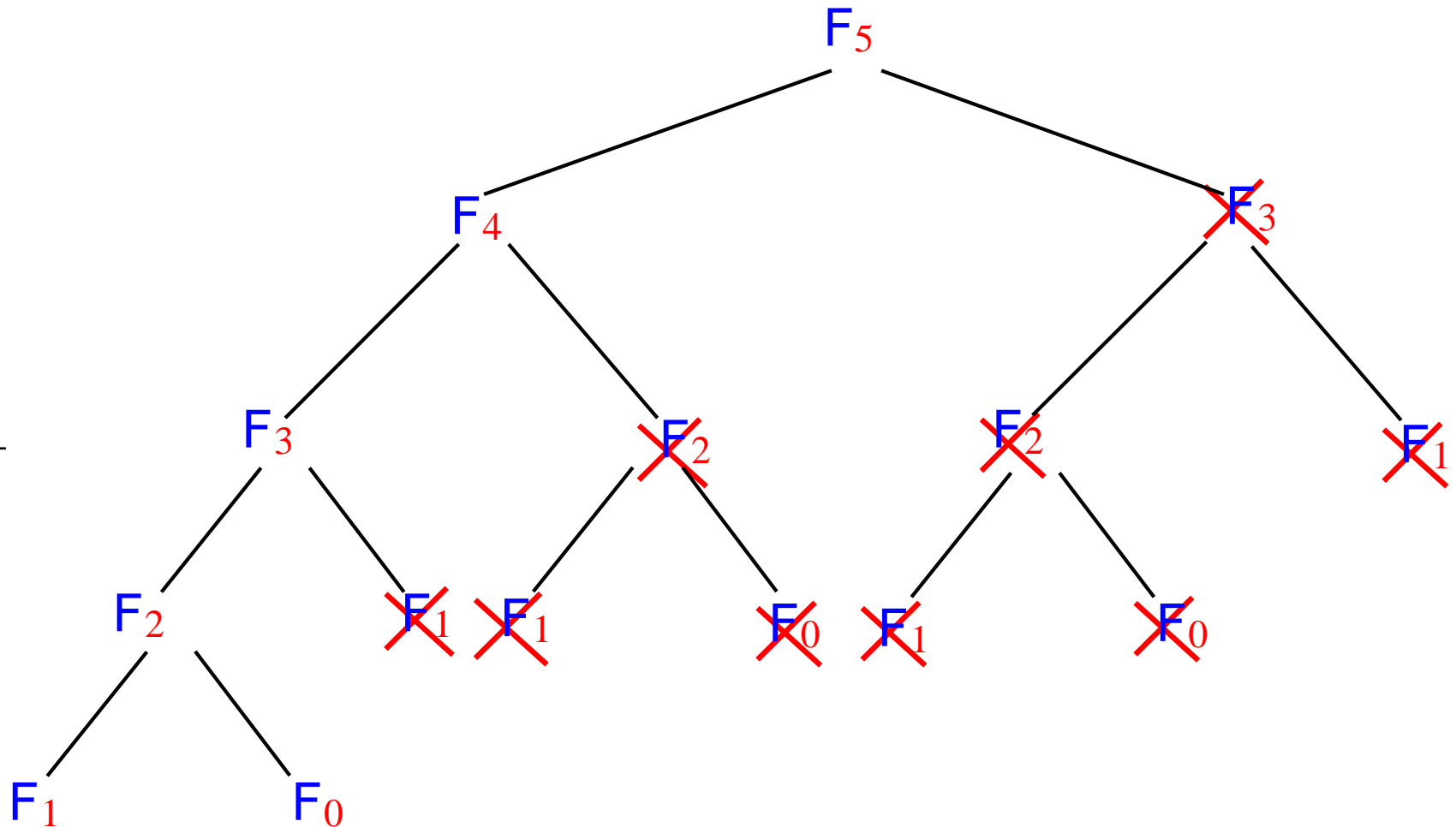
```
1  se  $f[n] \geq 0$ 
2      então devolva  $f[n]$ 
3  se  $n \leq 1$ 
4      então  $f[n] \leftarrow n$ 
5      senão  $f[n] \leftarrow$  LOOKUP-FIBO( $f, n - 1$ )
                        + LOOKUP-FIBO( $f, n - 2$ )
6  devolva  $f[n]$ 
```

Não recalcula valores de f .

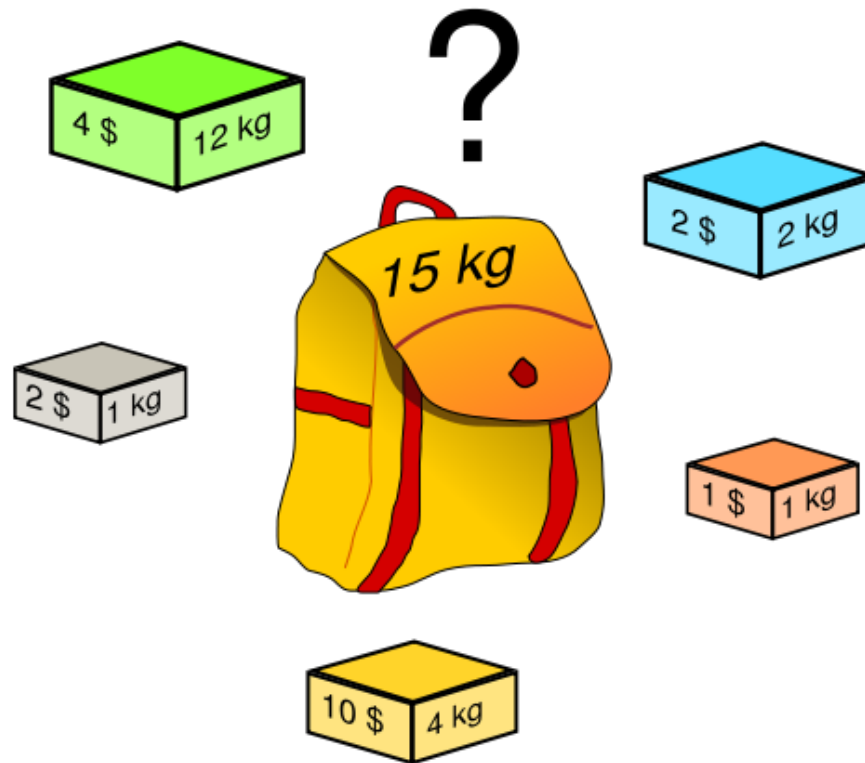
Consumo de tempo

Consumo de tempo é $\Theta(n)$.

Algoritmo resolve cada subproblemas apenas uma vez.



Knapsack[Problema da Mochila]



Knapsack[Problema da Mochila]

- Busca calcular a melhor maneira de se armazenar em um compartimento qualquer muitos itens com valores agregados, de forma que o somatório dos valores de tais itens seja o máximo possível dentre as possibilidades de combinação de itens.
- Usual em supermercados e empresas de armazenagem e logística em geral.