Programação dinâmica

CLRS cap 15

= "recursão–com–tabela"
= transformação inteligente de recursão em iteração

Números de Fibonacci

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

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 ≤ 1
2      então devolva n
3      senão a ← FIBO-REC (n − 1)
4            b ← FIBO-REC (n − 2)
5            devolva a + b
```


Consumo de tempo

```
FIBO-REC (n)
1  se n ≤ 1
2      então devolva n
3      senão a ← FIBO-REC (n − 1)
4            b ← FIBO-REC (n − 2)
5            devolva a + b
```

Tempo em segundos:

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

$F_{47} = 2971215073$

Consumo de tempo

```
FIBO-REC (n)
1  se n ≤ 1
2      então devolva n
3      senão a ← FIBO-REC (n − 1)
4            b ← FIBO-REC (n − 2)
5            devolva a+b
```

$T(n)$:= número de somas feitas por FIBO-REC (n)

linha	número de somas
1-2	$= 0$
3	$= T(n-1)$
4	$= T(n-2)$
5	$= 1$
$T(n)$	$= T(n-1) + T(n-2) + 1$

Recorrência

$$T(0) = 0$$
$$T(1) = 0$$
$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{para } n = 2, 3, \ldots$$

A que classe Ω pertence $T(n)$?
A que classe O pertence $T(n)$?

Solução: $T(n) > (3/2)^n$ para $n \geq 6$.

n	0	1	2	3	4	5	6	7	8	9
T_n	0	0	1	2	4	7	12	20	33	54
$(3/2)^n$	1	1.5	2.25	3.38	5.06	7.59	11.39	17.09	25.63	38.44

Recorrência

Prova: $T(6) = 12 > 11.40 > (3/2)^6$ e $T(7) = 20 > 18 > (3/2)^7$.
Se $n \geq 8$, então

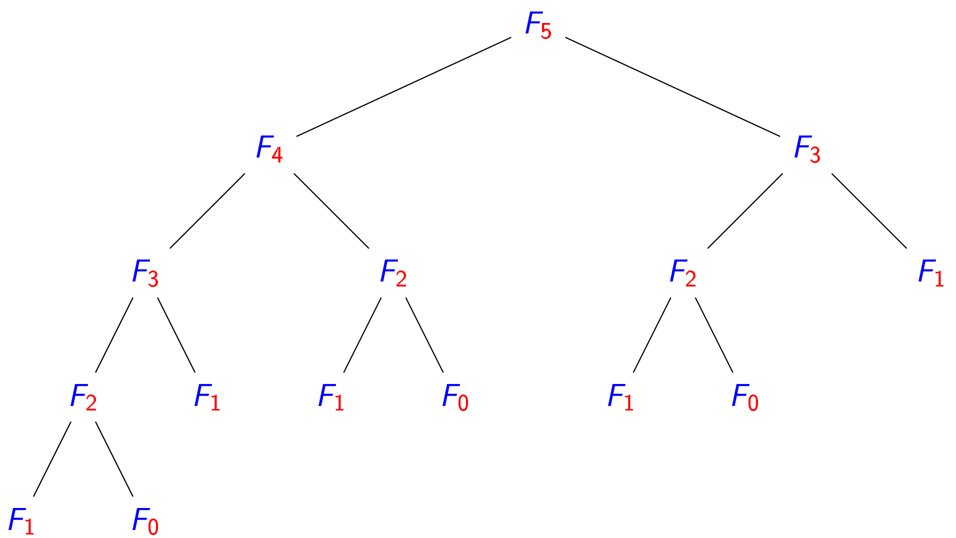
$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&\stackrel{\text{hi}}{>} (3/2)^{n-1} + (3/2)^{n-2} + 1 \\
&= (3/2 + 1)\,(3/2)^{n-2} + 1 \\
&> (5/2)\,(3/2)^{n-2} \\
&> (9/4)\,(3/2)^{n-2} \\
&= (3/2)^2 (3/2)^{n-2} \\
&= (3/2)^n .
\end{aligned}$$

Logo, $T(n)$ é $\Omega((3/2)^n)$. Verifique que $T(n)$ é $\mathrm{O}(2^n)$.

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)
```


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.

Versão recursiva com memoização

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

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

Não recalcula valores de f.

Algoritmo de programação dinâmica

Sem recursão:

FIBO (n)
$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
para $i \leftarrow 2$ **até** n **faça**
    $f[i] \leftarrow f[i-1] + f[i-2]$
devolva $f[n]$

Note a tabela $f[0\,..\,n{-}1]$.

f					$\star$	$\star$	??			

Consumo de tempo (e de espaço) é $\Theta(n)$.

Algoritmo de programação dinâmica

Versão com economia de espaço.

```
FIBO (n)
se n = 0 então devolva 0
f_ant ← 0
f_atual ← 1
para i ← 2 até n faça
    f_prox ← f_atual + f_ant
    f_ant ← f_atual
    f_atual ← f_prox
devolva f_atual
```

Consumo de tempo é $\Theta(n)$.

Consumo de espaço é $\Theta(1)$.

Corte de hastes

Hastes de aço-5711 são vendidas em pedaços de tamanho inteiro. As usinas produzem hastes longas, e os comerciantes cortam em pedaços para vender.

Por razões inexplicáveis, o preço de uma haste de tamanho i está tabelado como q_i, e para o problema abaixo ter graça, é bom que isso não seja função linear de i.

Problema: Dada uma haste de tamanho n e a tabela de preços p_i, qual a melhor forma de cortar para maximizar o preço de venda total?

Versão simplificada: qual o maior valor q_n que se pode obter de uma haste de tamanho n?

Solução recursiva

Corta-se um primeiro pedaço de tamanho i. e o pedaço restante, de tamanho $n - i$ do melhor jeito possível. O valor desse corte é

$$p_i + q_{n-i}$$

A questão é escolher o melhor i; é o que maximiza a expressão acima:

$$q_n = \max_{1 \leq i \leq n} p_i + q_{n-i}.$$

$$q_0 = 0.$$

Primeiro código

```
CORTA-HASTE (p, n)
se n == 0
    então devolva 0
q = −∞
para i = 1 até n
    q = max(q, p[i] + CORTA-HASTE(p, n − i))
devolva q
```

Tempo:

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

$$= 2^n.$$

Com memoização

Note que p e r funcionam como variáveis globais.

```
CORTA-HASTE-MEMOIZADO (p, n)
declare r[1 . . . n]
r[0] = 0
para i = 1 até n
    r[i] = −∞
devolva CORTA-HASTE-MEMOIZADO-AUX (p, n, r)
```

```
CORTA-HASTE-MEMOIZADO-AUX (p, n, r)
se r[n] ≥ 0
    devolva r[n]
senão q = −∞
    para i = 1 até n
        q = max(q, p[i] + CORTA-HASTE-MEMOIZADO-AUX(p, n − i, r))
    r[n] = q
    devolva q
```


Bottom Up

```
CORTA-HASTE-BOTTOM-UP (n)
declare r[1 . . . n]
r[0] = 0
para j = 1 até n
    q = −∞
    para i = 1 até j
        q = max(q, p[i] + r[j − i])
    r[j] = q
devolva q
```


Recuperando o melhor corte

```
CORTA-HASTE-BOTTOM-UP-COMPLETO (n)
declare r[1 . . . n], d[1 . . . n]
r[0] = 0
para j = 1 até n
    q = −∞
    para i = 1 até j
        se q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            d[j] = i
    r[j] = q
devolva q e d
```


Listando os cortes

(usando concatenação de listas, estilo python)

```
LISTA-CORTES(d, n)
se n == 0
    devolva [ ]
senão
    devolva [ d [n] ] .LISTA-CORTES(d, n − d[n])
```