Projeto e Análise de Algoritmos

Prof.Antonio Carlos Sobieranski

DEC7536 | ENC | DEC | CTS

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Programação Dinâmica

A Programação Dinâmica, semelhante a "Dividir para Conquistar", resolve problemas combinando as soluções de subproblemas.

- A técnica Dividir para Conquistar faz,
  1. Divide o problema original em subproblemas
  2. Resolve os subproblemas recursivamente
  3. Combina as soluções dos subproblemas para resolver o problema original
- Em contraste, a *Programação Dinâmica* é aplicada quando os subproblemas têm sobreposição.
  - Neste contexto, "Dividir para Conquistar" estaria realizando mais trabalho do que o necessário com as partes em comum dos subproblemas.

Programação Dinâmica

Exemplo Fibonacci Recursivo

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

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

$$Fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{se } n > 1 \end{cases}$$

Programação Dinâmica

Exemplo Fibonacci Recursivo

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

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

```
Fib(5)
 ├─Fib(3)
 │  ├─Fib(1)
 │  └─Fib(2)
 │     ├─Fib(0)
 │     └─Fib(1)
 └─Fib(4)
    ├─Fib(2)
    │  ├─Fib(0)
    │  └─Fib(1)
    └─Fib(3)
       ├─Fib(1)
       └─Fib(2)
          ├─Fib(0)
          └─Fib(1)
```

Programação Dinâmica

Exemplo Fibonacci Recursivo – Consumo de Tempo

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

$F_{47} = 2971215073$

Programação Dinâmica

Exemplo Fibonacci Recursivo – Consumo de Tempo

Algoritmo Recursivo* X Iterativo

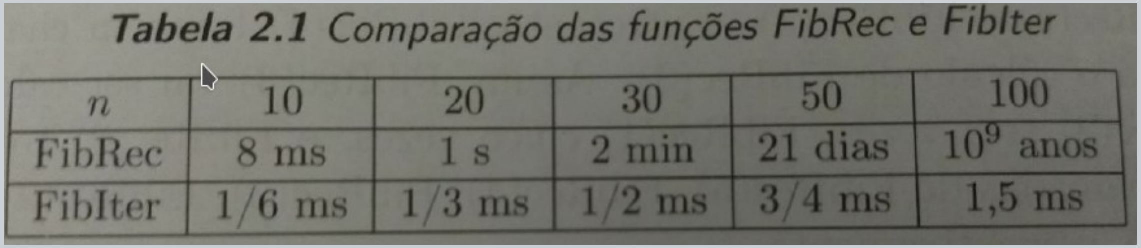


Tabela 2.1 Comparação das funções FibRec e FibIter

n	10	20	30	50	100
FibRec	8 ms	1 s	2 min	21 dias	10^9 anos
FibIter	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

```c
static int acumR = 0;

unsigned int fiboR(unsigned int n)
{
    acumR++;

    if(n == 0 || n == 1) return n;
    else return fiboR(n-2)+fiboR(n-1);
}

unsigned int fiboI(unsigned int n)
{
    unsigned int value = 0;
    if(n == 0) return 0;
    if(n == 1) return 1;

    unsigned int V1 = 0;
    unsigned int V2 = 1;

    for(int i=1; i<=n; i++)
    {
            value = V1+V2;
            V2 = V1;
            V1 = value;
    }
    return value;
}

int main()
{
    unsigned int fiborec = fiboR(20);
    unsigned int fiboiter = fiboI(20);


    printf("%d - acum: %d\n", fiborec, acumR);
    printf("%d\n", fiboiter);
    return 0;
}
```

Quais as complexidades computacionais de **tempo** e **espaço** para cada algoritmo ?

**Outras constantes de tempo consideradas comparado slide anterior*

Programação Dinâmica

Exemplo Fibonacci Recursivo – Consumo de Tempo

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

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

$$F_{47} = 2971215073$$

$$f_n = \frac{1}{\sqrt{5}}[\Phi^n - (-\Phi)^{-n}],$$

Programação Dinâmica

Exemplo Fibonacci Recursivo – 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** $a+b$

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

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

Programação Dinâmica

Exemplo Fibonacci Recursivo – 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 $T(n)$?

Programação Dinâmica

Exemplo Fibonacci Recursivo – 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 $n \geq 6$.

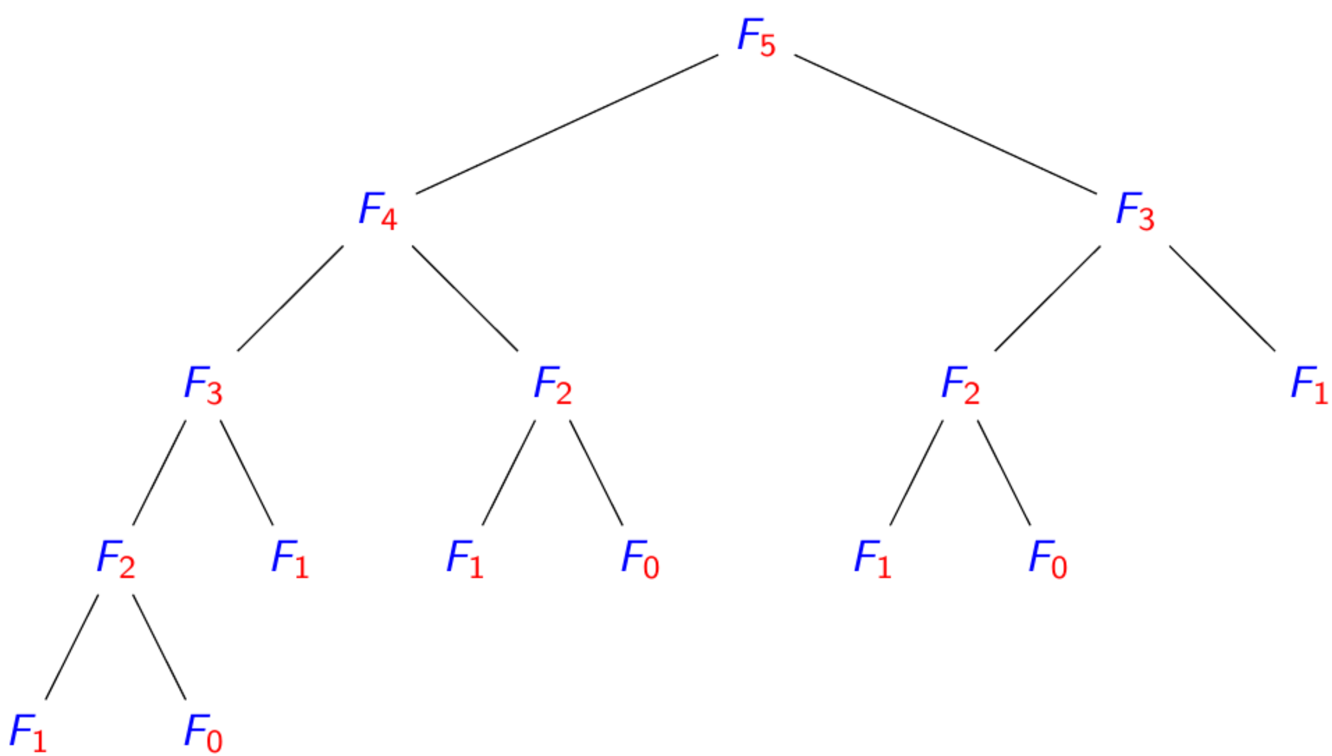
n	0	1	2	3	4	5	6	7	8	9
T_n	0	0	1	2	4	7	12	20	33	54
$(3/2)^n$	1	1.5	2.25	3.38	5.06	7.59	11.39	17.09	25.63	38.44

Programação Dinâmica

Exemplo Fibonacci Recursivo – Recorrência

Consumo de tempo é exponencial.

Algoritmo resolve subproblemas muitas vezes.



Programação Dinâmica

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

Programação Dinâmica

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

Responsa Rapidamente:

Existe uma solução com complexidade de tempo menor que *O(n)* para os números de Fibonacci.

Show me !

```c
static int acumR = 0;

unsigned int fiboR(unsigned int n)
{
    acumR++;

    if(n == 0 || n == 1) return n;
    else return fiboR(n-2)+fiboR(n-1);
}

unsigned int fiboI(unsigned int n)
{
    unsigned int value = 0;
    if(n == 0) return 0;
    if(n == 1) return 1;

    unsigned int V1 = 0;
    unsigned int V2 = 1;

    for(int i=1; i<=n; i++)
    {
            value = V1+V2;
            V2 = V1;
            V1 = value;
    }
    return value;
}

int main()
{
    unsigned int fiborec = fiboR(20);
    unsigned int fiboiter = fiboI(20);


    printf("%d - acum: %d\n", fiborec, acumR);
    printf("%d\n", fiboiter);
    return 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.

Programação Dinâmica

Um algoritmo de programação dinâmica resolve cada subproblema e salva sua solução em uma tabela.

- Evitando o trabalho de recomputar uma resposta toda vez que um subproblema for resolvido.

A programação dinâmica é tipicamente aplicada à **Problemas de Otimização**.

- Cada subproblema tem muitas soluções possíveis.
- Cada solução tem um *valor*. Deseja-se encontrar a solução de valor ótimo (máximo ou mínimo).

Programação Dinâmica

Ao se desenvolver um algoritmo em programação dinâmica, segue-se quatro passos:

1. Caracterização de estrutura de uma solução ótima.
2. Recursivamente define-se o valor de uma solução ótima.
3. Computa-se o valor da solução ótima, tipicamente *bottom-up*.
4. Constrói-se uma solução ótima a partir da informação computada.

Os passos de 1 ao 3 geram a base da solução de um problema por programação dinâmica.

Programação Dinâmica

FIBO-REC (n)

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

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.

Programação Dinâmica

Sem recursão:

```
FIBO (n)
f[0] ← 0
f[1] ← 1
para i ← 2 até n faça
    f[i] ← 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 $\Theta(n)$.

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 $\Theta(1)$.

Problema do Corte da Haste

Vamos tentar resolver o problema do Corte de uma Haste.

- Imagine que uma empresa compra longas haste de aço e as corta em hastes menores, vendendo-as.
- Cada corte é livre, podendo as hastes resultantes terem qualquer tamanho menor que a haste original.
  - Desta forma as sub-hastes poderão ter preço de venda diferentes.
- A empresa deseja saber qual a melhor forma de realizar estes cortes.
  - Suponha que seja conhecido o preço p_i de cada haste com o comprimento em $i = 1, 2, 3, \ldots$ polegadas.
  - Suponha também que cada haste sempre terá um número inteiro de polegadas.

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	8	9	10	17	17	20	24	30

Problema do Corte da Haste

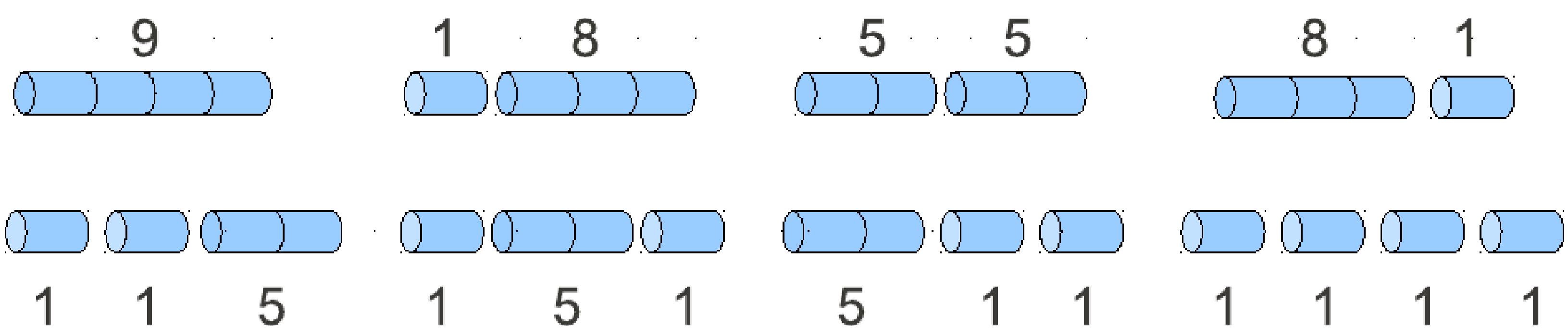
Problema do Corte da Haste

Dada uma haste de comprimento de n polegadas e dada uma tabela de preços p_i para $i = 1, 2, \ldots, n$, determine o valor máximo possível de receita r_n possível de obter com o corte desta haste e a venda das respectivas sub-hastes.

Note que se o valor p_n for grande o suficiente, não haverá nenhum corte!

Problema do Corte da Haste

- Suponha, como exemplo, o caso com $n = 4$ polegadas! Considerando a possibilidade de a cada polegada haja ou não um corte, para uma haste de n polegadas haverá 2^{n-1} formas de executar os cortes.
  - Observe que irão existir situações equivalentes, visto que não há distinção entre os cortes. Caso seja considerado apenas os casos não redundantes, a função de *quantidade de cortes* é chamada de **Função de Partição**.



Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	8	9	10	17	17	20	24	30

Problema do Corte da Haste

- Se uma solução ótima corta a haste em k pedaços $(1 \leqslant k \leqslant n)$, então uma decomposição ótima

$$n = i_1 + i_2 + \cdots + i_k$$

da haste em padaços de comprimento $i_1, i_2, \ldots, i_k$ irá prover a receita máxima

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

- Para o problema descrito pela tabela,

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	8	9	10	17	17	20	24	30

é possível determinar por inspeção as receitas r_i, para $i = 1, 2, \ldots, 10$.

Problema do Corte da Haste

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	8	9	10	17	17	20	24	30

Recita	Decrição
$r_1 = 1$	solução $1 = 1$ sem corte
$r_2 = 5$	solução $2 = 2$ sem corte
$r_3 = 8$	solução $3 = 3$ sem corte
$r_4 = 10$	solução $4 = 2 + 2$
$r_5 = 13$	solução $5 = 2 + 3$
$r_6 = 17$	solução $6 = 6$ sem corte
$r_7 = 18$	solução $7 = 1 + 6$ ou $7 = 2 + 2 + 3$
$r_8 = 22$	solução $8 = 6 + 2$
$r_9 = 25$	solução $9 = 3 + 6$
$r_{10} = 30$	solução $10 = 10$ sem corte

Sub-estrutura ótima !!!

Problema do Corte da Haste

- Observe que é possível descrever o problema de forma genérica, maximizando r_n para $n \geqslant 1$ em termos das receitas ótimas de hastes mais curtas,
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \tag{1}$$
- Desta forma, para resolver o problema original de tamanho n, são resolvidos problemas do mesmo tipo, porém com tamanhos menores.
- Uma vez realizado o primeiro corte, é possível encarar o problema com duas instâncias independentes do mesmo problema.
- A solução ótima completa incorpora as solução ótimas dos dois sub-problemas.
- É dito que o Problema do Corte da Haste exibe uma **Subestrutura Ótima**
  - A solução ótima do problema incorpora as soluções ótimas dos subproblemas, os quais são resolvidos independentemente.

Problema do Corte da Haste

- O problema pode ser visto como a divisão da haste em dois pedaços,
  1. O primeiro pedaço de comprimento i, e
  2. O segundo pedaço de comprimento $n - i$.
- Observe que apenas o segundo pedaço poderá ser novamente dividido.
- Assim, o problema pode ser visto como: Um primeiro pedaço seguido de uma decomposição do segundo pedaço.
- É possível escrever uma simplificação da Equação 1,

$$r_n = \max_{1 \leqslant i \leqslant n} (p_i + r_{n-i}) \tag{2}$$

onde $r_0 = 0$.

Problema do Corte da Haste

O seguinte pseudo-código implementa a computação da Equação 2, de uma forma recursiva,

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

- $p \rightarrow p[1..n]$ é um arranjo de preços e n é um número inteiro.
- O retorno será a receita máxima possível para uma haste de comprimento n

Problema do Corte da Haste

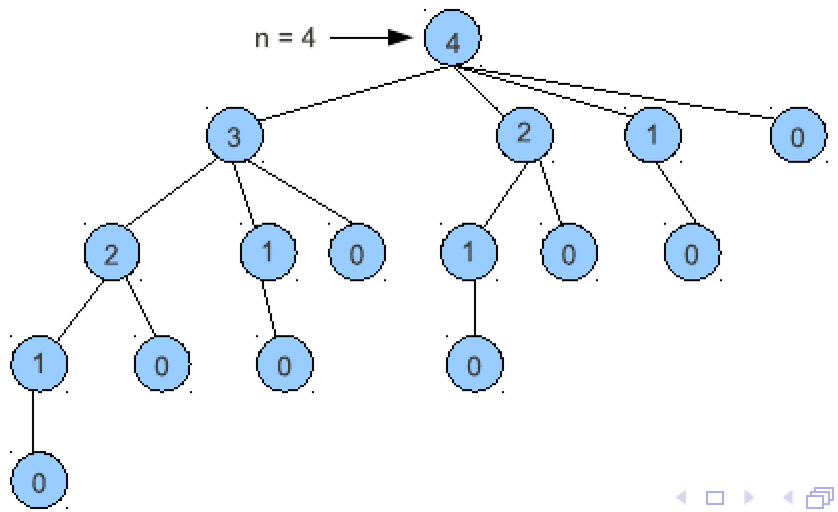
Demonstração do Algoritmo Recursivo

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Comprimento i	1	2	3	4	5	6	7	8	9	10
Preço p_i	1	5	8	9	10	17	17	20	24	30

Problema do Corte da Haste

- Ao implementar este pseudo-código em alguma linguagem de programação é possível observar que para tamanhos moderadamente grandes de n, o programa demorará um tempo muito longo para ser executado.
- Por que o código CUT-ROD é ineficiente?
  - Da forma com que está sendo implementado, CUT-ROD invoca recursivamente vários problemas idênticos.



Problema do Corte da Haste

- Seja $T(n)$ o número total de chamadas feitas pelo CUT-ROD quando o segundo argumento é n.
  - Desta forma, $T(0) = 1$ e

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \tag{3}$$

  o que leva a solução,

$$T(n) = 2^n$$

Problema do Corte da Haste

Aplicando Programação Dinâmica

- A solução inicial resolve o mesmo subproblema várias vezes!
- É desejado que cada subproblema seja resolvido apenas uma vez, salvando-a
- Se um subproblema for requerido ser resolvido mais de uma vez, deve-se "ler" a solução já salva.
- A programação dinâmica usa memória adicional para reduzir o tempo de processamento computacional.
  - Um custo exponencial no tempo pode ser transformado em um custo polinomial.
  - Isto irá ocorrer quando o número de subproblemas a serem resolvidos cresce de forma polinomial e o custo em tempo para resolver cada um deste subproblemas também é polinomial.

Problema do Corte da Haste

Aplicando Programação Dinâmica

- De foram geral, há duas forma de implementar a programação dinâmica:
  1. **Top-down with memoization**
     - Esta é a forma recursiva natural do problema, com a modificação de salvar o resultado de cada subproblema.
     - Antes de resolver um subproblema, verifica se a solução já foi computada.
  2. **Bottom-up**
     - Esta forma necessita definir o tamanho de um subproblema.
     - Esta ordena os subproblemas por seu tamanho, resolvendo do menor para o maior, de tal forma que aundo um dado subproblema é resolvido, todos os subproblemas menores já foram resolvidos.

Problema do Corte da Haste

Top-down com Memoization

```
MEMOIZED-CUT-ROD(p,n)
let r[0..n] be a new array — all elements are −∞
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p,n,r)
```

```
MEMOIZED-CUT-ROD-AUX(p,n,r)
if r[n] ⩾ 0
        return r[n]
if n == 0
        q = 0
else q = −∞
        for i = 1 to n
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Problema do Corte da Haste

Bottom-up

```
    BOTTOM-UP-CUT-ROD(p,n)
1   let r[0..n] be a new array
2   r[0] = 0
3   for j = 1 to n
4       q = −∞
5       for i = 1 to j
6           q = max(q, p[i] + r[j − i])
7       r[j] = q
8   return r[n]
```

- Um subproblema de tamanho i é menor que um de tamanho j, se $i < j$.
- O procedimento resolve problemas $j = 0, 1, \ldots, n$, nesta ordem.
- $r[0] = 0$, haste de tamanho 0 gera receita 0.
- Das linas $3 - 6$ o procedimento resolve os problemas de tamanho j.
- A linha 7 salva a solução do problema j
- A linha 8 retorna $r[n]$ que é o valor ótimo de r_n.

Problema do Corte da Haste

Bottom-up – Análise do *f(n)* do algoritmo

```
BOTTOM-UP-CUT-ROD(p,n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Problema do Corte da Haste

Top-down X Bottom-up com Programação Dinâmica

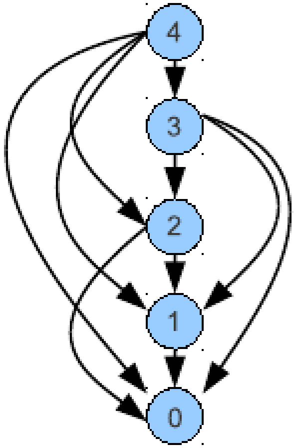
- Ambas as versão *bottom-up top-down* têm o mesmo custo em tempo assintótico de $\Theta(n^2)$.
- No pseudo-código da versão *bottom-up*, devido aos laços aninhados, é fácil verificar $\Theta(n^2)$.
- No pseudo-código da versão *top-down* não é tão direto devido a recursividade! (Como verificar?)

Problema do Corte da Haste

Grafo dos sub-problemas

Ao se pensar em um problema de programação dinâmica, pensa-se em um conjunto de subproblemas acoplados e como é este acoplamento, ou seja, como um subproblema depende de outro subproblemas.

- Um grado dos subproblemas incorpora esta informação.
- Pense no problema do corte da haste. Se $n = 4$, o grafo dos subproblemas será,



Problema do Corte da Haste

Reconstruindo uma solução para o problema do corte

- Observe que a solução da programação dinâmica para o problema do corte da haste nos dá o valor da solução ótima!
  - Porém, não nos informa a lista dos tamanhos dos pedaços a serem cortados!
- Contudo, é possível ainda definir uma *escolha* no algoritmo que nos conduza a solução ótima.
  - É possível estender o procedimento BOTTOM-UP-CUT-ROD para computar para cada haste de comprimento j não somente a receita ótima r_j, mas também o tamanho ótimo do primeiro pedaço a ser cortado.

Problema do Corte da Haste

Reconstruindo uma solução para o problema do corte

```
PRINT-CUT-ROD-SOLUTION(p,n)
(r, s) =EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
While n > 0
    print s[n]
   n = n − s[n]
```

```
EXETENDED-BOTTOM-UP-CUT-ROD(p,n)
let r[0..n] e s[0..n] be new arrays
r[0] = 0
for j = 1 to n
   q = −∞
    for i = 1 to j
       if q < p[i] + r[j − i]
          q = p[i] + r[j − i]
          s[j] = i
   r[j] = q
return r and s
```

Programação Dinâmica

Sugestão de Leitura

- Capítulo 15, seção 15.1 do lívro do Cormen;
- Richard Bellman. *Dynamic Programming*. Princeton Univerty Press. 1957.
- Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49-76, 1992.

Contato

Prof.Antonio Carlos Sobieranski

DEC | C112

E-mail: a.sobieranski@ufsc.br

Agradecimentos ao Prof.Thiago Ferreira por alguns slides utilizados nesta aula

UNIVERSIDADE FEDERAL
DE SANTA CATARINA