

PROGRAMAÇÃO DINÂMICA

MO417 - Complexidade de
Algoritmos I

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

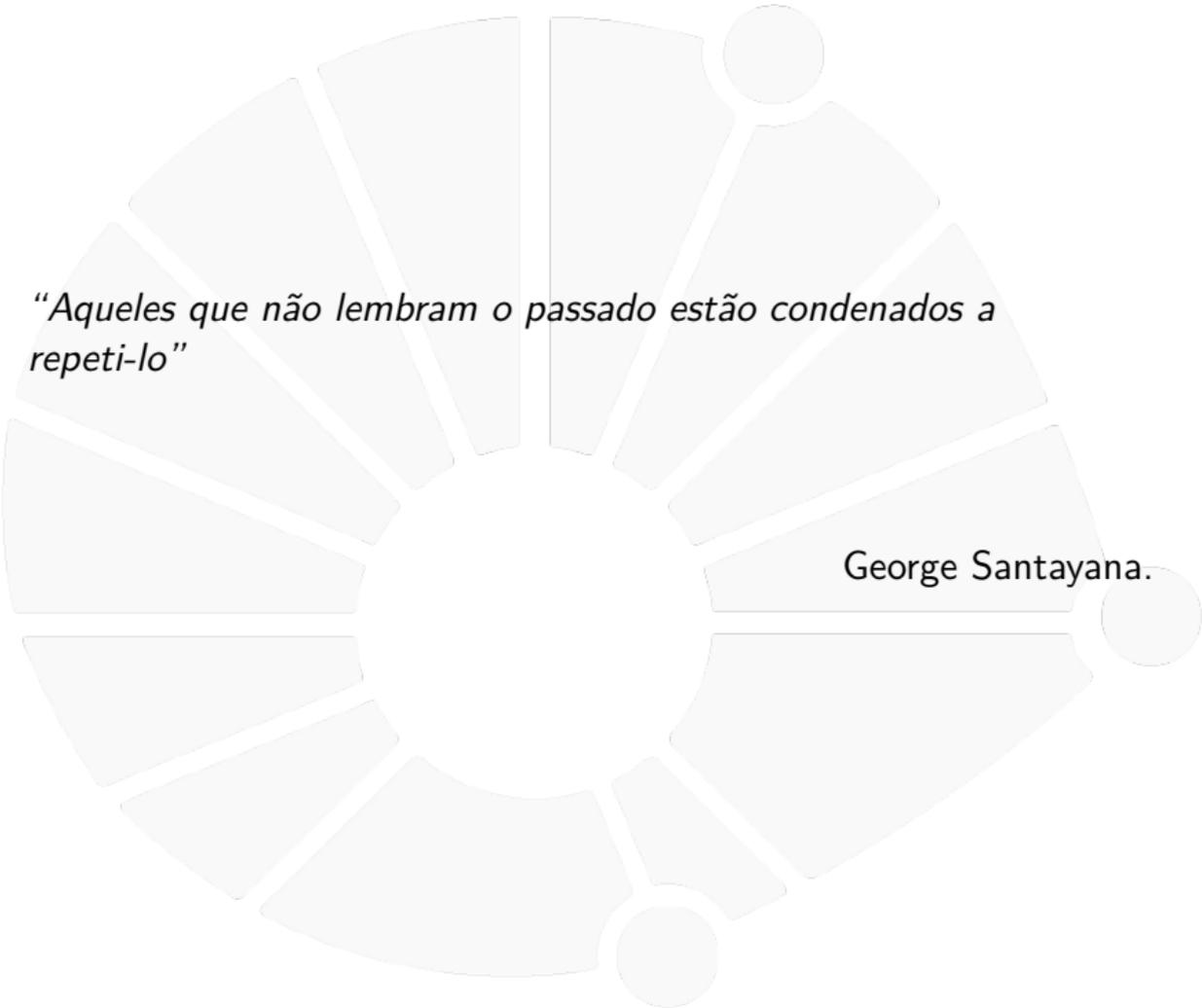
04/24

13



UNICAMP



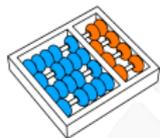


“Aqueles que não lembram o passado estão condenados a repeti-lo”

George Santayana.



INTRODUÇÃO



Richard Ernest Bellman (26/08/1920 – 19/03/1984)



Principais linhas de atuação:

- ▶ Matemática e teoria de controle.
- ▶ Equações diferenciais.
- ▶ Programação dinâmica e estocástica.
- ▶ Algoritmos em grafos.

Prêmios:

- ▶ Prêmio Dickson de Ciências (1970).
- ▶ Prêmio Norbert Wiener (1970).
- ▶ Prêmio Teoria John von Neumann (1976).
- ▶ Medalha de Honra IEEE (1979).
- ▶ Prêmio Richard E. Bellman (1984).



Problemas e subproblemas

Vamos estudar problemas com algumas propriedades:

▶ **Subestrutura ótima:**

- ▶ Decompomos a instância em vários **subproblemas**.
- ▶ A solução ótima é construída a partir de soluções ótimas de subproblemas.

▶ **Sobreposição de subproblemas:**

- ▶ Uma instância é quebrada em várias instâncias menores.
- ▶ A recursão recalcula uma mesma instância **várias vezes**.



Programação dinâmica

Ideia:

- ▶ Evitar o recálculo desnecessário de subproblemas.
- ▶ Guardar as soluções de subproblemas em uma **tabela**.
- ▶ Cada entrada é uma instância distinta do subproblema.

Premissas da programação dinâmica:

- ▶ O número de entradas da tabela é pequeno.
- ▶ Sabemos computar cada uma eficientemente.

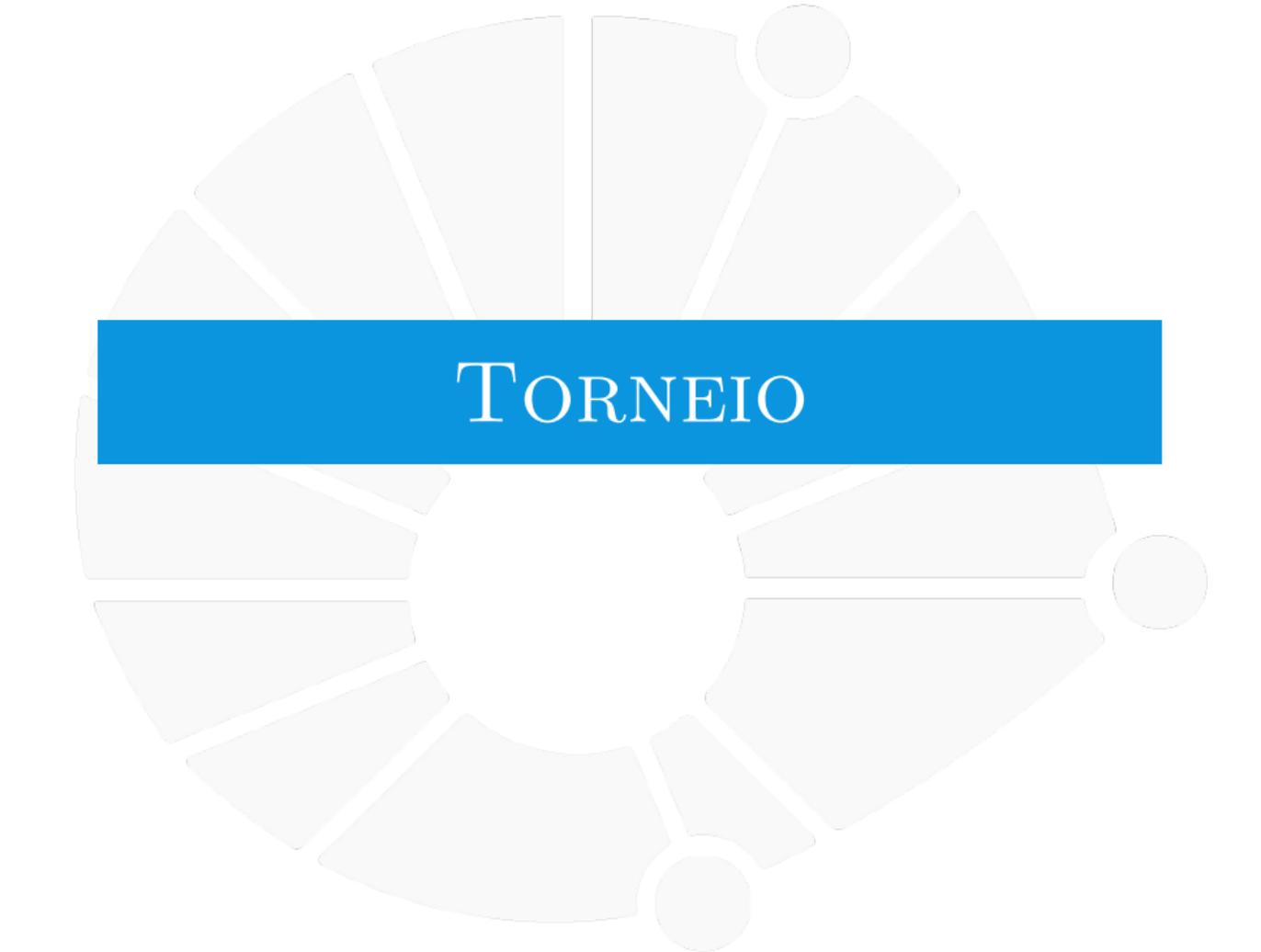


Problemas de otimização

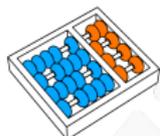
Consideramos tipicamente **problemas de otimização**:

- ▶ Cada instância tem um conjunto de **soluções viáveis**.
- ▶ Cada uma delas tem um valor dado pela **função-objetivo**.
- ▶ Queremos alguma cujo valor é **mínimo ou máximo**.

Uma solução viável com o melhor valor é chamada de **ÓTIMA**.



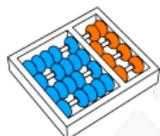
TORNEIO



Problema do torneio

- ▶ Considere um torneio entre duas pessoas A e B .
- ▶ Nesse jogo, uma partida nunca termina empatada.
- ▶ Vence o torneio o jogador que vencer k partidas primeiro.
- ▶ Suponha que A vence uma partida com probabilidade 0.6.
 - ▶ Outros valores também poderiam ser usados.

Problema: Qual é a probabilidade de que A vença o torneio?

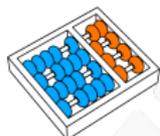


Subproblema recursivo

Seja $P(k|i, j)$ a probabilidade de A ganhar o torneio dado que:

- ▶ O número de vitórias do jogador A é i .
- ▶ O número de vitórias do jogador B é j .

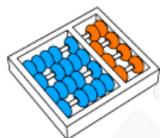
Vamos construir um algoritmo por **indução reversa** em i e em j .



Algoritmo por indução reversa

- ▶ Base:
 - ▶ Se $i = k$, então $P(k|k, j) = 1$ para todo j pois A é o vencedor.
 - ▶ Se $j = k$, então $P(k|i, k) = 0$ para todo i pois B é o vencedor.
- ▶ Hipótese de indução:
 - ▶ Sabemos calcular $P(k|i', j')$ para $i' > i$ e $j' > j$.
- ▶ Passo:
 - ▶ Considere os casos em que A ou B vence a próxima partida.
 - ▶ Por hipótese, já sabemos calcular $P(k|i + 1, j)$ e $P(k|i, j + 1)$.
 - ▶ Então podemos calcular a probabilidade condicional:

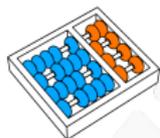
$$P(k|i, j) = 0.6 \cdot P(k|i + 1, j) + 0.4 \cdot P(k|i, j + 1)$$



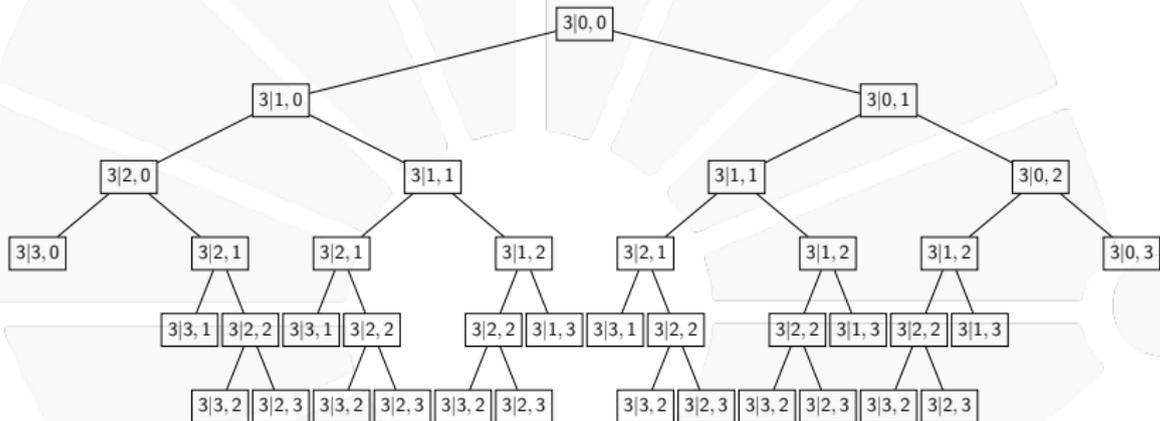
Algoritmo recursivo

Algoritmo: RECURSIVO(k, i, j)

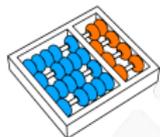
- 1 se $i = k$
 - 2 └ devolva 1
 - 3 se $j = k$
 - 4 └ devolva 0
 - 5 devolva
 $0.6 \cdot \text{RECURSIVO}(k, i + 1, j) + 0.4 \cdot \text{RECURSIVO}(k, i, j + 1)$
-



Esboço da árvore de recursão para $k = 3$

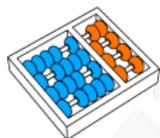


- ▶ A menor profundidade de uma folha é k .
- ▶ Esta árvore tem pelo menos $\sum_{p=0}^k 2^p = 2^{k+1} - 1$ nós.
- ▶ O algoritmo recursivo é $\Omega(2^k)$.

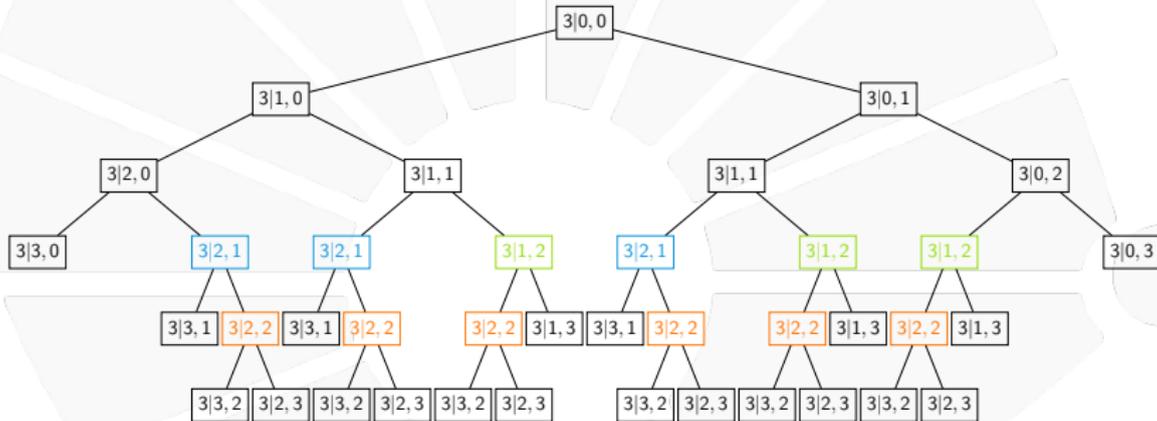


Número de subproblemas

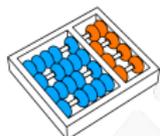
- ▶ Temos um algoritmo exponencial, que tem pouca utilidade.
- ▶ Mas os parâmetros de entrada i e j variam apenas 0 até k .
- ▶ O número de possibilidades é $(k + 1)(k + 1)$.
- ▶ Ou seja, há apenas $O(k^2)$ instâncias do subproblema.



Número de subproblemas



- ▶ Vários subproblemas aparecem repetidas vezes.
- ▶ Alguns deles estão com a mesma cor na árvore acima.



Duas estratégias

1. Top-down:

- ▶ Resolvemos os subproblemas novos recursivamente.
- ▶ Guardamos os resultados em uma estrutura de dados.
- ▶ É chamada de programação dinâmica com memorização ou simplesmente de **memorização**.

2. Bottom-up:

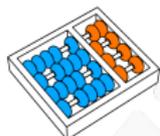
- ▶ Criamos uma tabela para guardar os resultados.
- ▶ Preenchemos as entradas para os casos básicos.
- ▶ Preenchemos as demais, das menores para as maiores.
- ▶ É chamada tipicamente de **programação dinâmica**.



Exemplo da estratégia bottom-up

	0	1	2	3
0				
1				
2				
3				

- ▶ Criamos uma matriz M de tamanho $(k + 1) \times (k + 1)$.
- ▶ Cada posição $M[i, j]$ guarda o valor de $P(k|i, j)$.
- ▶ O índice da linha representa i e da coluna j .
- ▶ Por exemplo, para $k = 3$ temos uma matriz 4×4 .



Casos básicos

	0	1	2	3
0				0
1				0
2				0
3	1	1	1	-

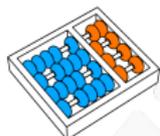
- ▶ Inicializamos as entradas para os casos básicos.
- ▶ Fazemos $M[k, j] = 1$ e $M[i, k] = 0$.



Passo

	0	1	2	3
0	0.68	0.48	0.22	0
1	0.82	0.65	0.36	0
2	0.94	0.84	0.6	0
3	1	1	1	-

- ▶ Uma entrada vale $M[i, j] = 0.6 \cdot M[i + 1, j] + 0.4 \cdot M[i, j + 1]$.
- ▶ Cada uma das entradas depende dos vizinhos na próxima linha $M[i + 1, j]$ e na próxima coluna $M[i, j + 1]$.
- ▶ Preenchemos a partir das últimas linha e coluna.
- ▶ A resposta está em $M[0, 0]$.



Algoritmo com programação dinâmica

- ▶ Passamos a probabilidade de A ganhar como parâmetro g .

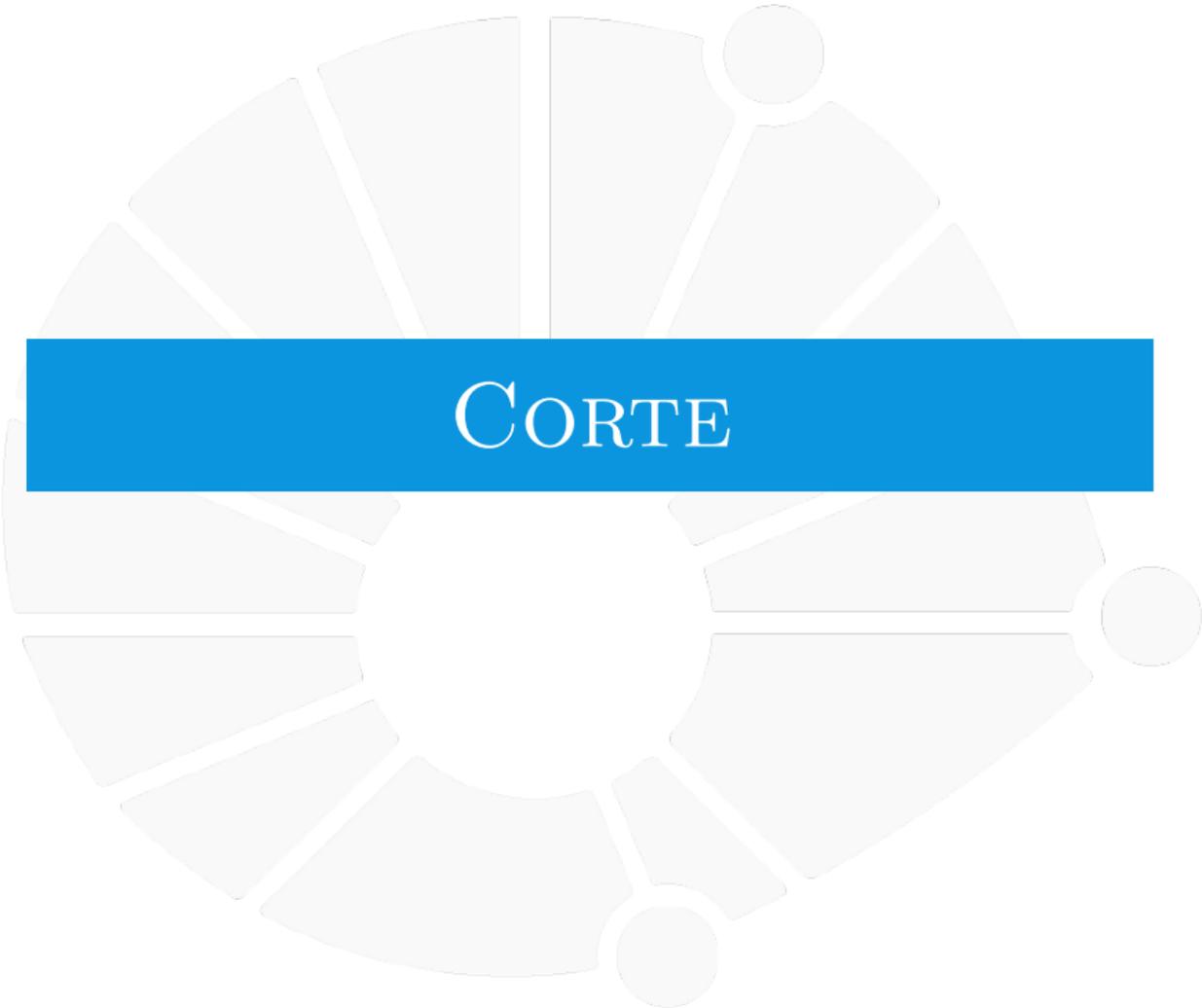
Algoritmo: $PD(k, g)$

```

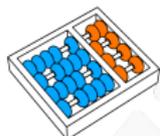
1 para  $i \leftarrow 0$  até  $k - 1$ 
2    $M[k, i] \leftarrow 1$ 
3    $M[i, k] \leftarrow 0$ 
4 para  $i \leftarrow k - 1$  até 0
5   para  $j \leftarrow k - 1$  até 0
6      $M[i, j] \leftarrow g \cdot M[i + 1, j] + (1 - g) \cdot M[i, j + 1]$ 
7 devolva  $M[0, 0]$ 

```

- ▶ A complexidade de tempo do algoritmo é $O(k^2)$.

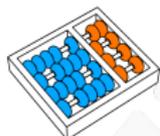


CORTE



Corte de barras de aço

- ▶ Uma empresa corta longas **barras de aço** em pedaços menores para revenda.
- ▶ Realizar um corte tem um custo insignificante.
- ▶ Uma barra tem tamanho inteiro n e podemos cortá-la em qualquer posição $1 \leq i \leq n - 1$ ou revender a barra inteira.
- ▶ Um pedaço de tamanho $1 \leq i \leq n$ tem preço de revenda p_i .



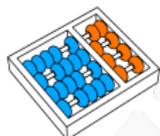
Exemplo de barra de aço

Exemplo onde $n = 4$:

tamanho	1	2	3	4
preço	1	5	8	9

Algumas formas de cortar:

- ▶ 4 itens de tamanho 1, **preço total 4.**
- ▶ 2 itens de tamanho 2, **preço total 10.**
- ▶ 2 itens de tamanho 1 e 1 item de tamanho 2, **preço total 7.**



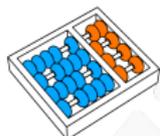
Problema do corte de barra de aço

Problema

Entrada: Um inteiro n e um vetor com os preços de revenda p .

Solução: Quais cortes realizar na barra.

Objetivo: **MAXIMIZAR** preço total de revenda dos pedaços.



Projeto por indução

Seja r_n o preço total ótimo para uma barra de tamanho n .

▶ Base:

- ▶ Se $n = 0$ então $r_0 = 0$.
- ▶ Se $n = 1$ então $r_n = p_1$.

▶ Hipótese de indução:

- ▶ Sabemos calcular r_k para $k < n$.

▶ Passo:

- ▶ Seja i o tamanho do primeiro pedaço da solução ótima.
- ▶ A melhor forma de cortar o resto da barra vale r_{n-i} , daí:

$$r_n = p_i + r_{n-i}$$

- ▶ Como não conhecemos o tamanho do primeiro pedaço:

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



Algoritmo recursivo

Algoritmo: RECURSIVO(p, n)

```

1 se  $n = 0$ 
2   └ devolva 0
3  $s \leftarrow 0$ 
4 para  $i \leftarrow 1$  até  $n$ 
5   └  $t \leftarrow p_i + \text{RECURSIVO}(p, n - i)$ 
6     └ se  $t > s$ 
7       └  $s \leftarrow t$ 
8 devolva  $s$ 

```

A complexidade de tempo do algoritmo é dada por:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ \sum_{j=0}^{n-1} T(j) + \Theta(1) & \text{se } n > 0 \end{cases}$$



Tempo do algoritmo recursivo

Subtraindo $T(n-1)$ de $T(n)$ obtemos:

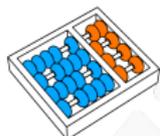
$$T(n) - T(n-1) = \left(\sum_{j=0}^{n-1} T(j) + c \right) - \left(\sum_{j=0}^{n-2} T(j) + c \right) = T(n-1)$$

Ou seja:

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

Resolvendo esta última recorrência, obtemos:

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



Algoritmo de PD

- ▶ Criamos vetor $r[0 \dots n]$ para usar programação dinâmica.
- ▶ Preenchemos do caso base para os casos maiores.

Algoritmo: $PD(p, n, r)$

```

1  $r[0] \leftarrow 0$ 
2 para  $n' \leftarrow 1$  até  $n$ 
3    $s \leftarrow 0$ 
4   para  $i \leftarrow 1$  até  $n'$ 
5      $t \leftarrow p_i + r[n' - i]$ 
6     se  $t > s$ 
7        $s \leftarrow t$ 
8    $r[n'] \leftarrow s$ 

```

- ▶ O algoritmo tem complexidade de tempo $\Theta(n^2)$.



MULTIPLICAÇÃO DE MATRIZES



Parentização

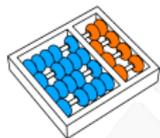
Considere um produto de matrizes:

$$M = M_1 \times M_2 \times \dots \times M_i \cdots \times M_n$$

- ▶ As dimensões das matrizes são dadas por um vetor b .
- ▶ A matriz M_i tem b_{i-1} linhas e b_i colunas.

Ordem de multiplicação:

- ▶ Só podemos multiplicar matrizes aos pares.
- ▶ Devemos escolher uma **parentização** para o produto.
- ▶ Produto matrizes $m \times l$ e $l \times n$ faz $m \cdot l \cdot n$ multiplicações.



Exemplo de parentização

Exemplo:

- ▶ Seja $M = M_1 \times M_2 \times M_3 \times M_4$ tal que $b = \langle 200, 2, 30, 20, 5 \rangle$.
- ▶ As possíveis parentizações são:

$M = (M_1 \times (M_2 \times (M_3 \times M_4)))$	→ 5.300 multiplicações
$M = (M_1 \times ((M_2 \times M_3) \times M_4))$	→ 3.400 multiplicações
$M = ((M_1 \times M_2) \times (M_3 \times M_4))$	→ 4.500 multiplicações
$M = ((M_1 \times (M_2 \times M_3)) \times M_4)$	→ 29.200 multiplicações
$M = (((M_1 \times M_2) \times M_3) \times M_4)$	→ 152.000 multiplicações

- ▶ A ordem das multiplicações faz diferença!



Multiplicação de cadeias de matrizes

Problema

Entrada: Um vetor b com dimensões de n matrizes.

Solução: Uma parentização das matrizes.

Objetivo: **MINIMIZAR** o número de multiplicações.

Testando todas as soluções viáveis:

- ▶ O número de parentizações é dado por:

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- ▶ A solução dessa recorrência é $P(n) = \Omega(4^n/n^{\frac{3}{2}})$.
- ▶ Um algoritmo de força bruta é **IMPRATICÁVEL**.



Encontrando uma subestrutura ótima

Considere uma **parentização ótima**.

- ▶ Para cada par (i, j) tal que $1 \leq i \leq j \leq n$, defina:

$$M_{i,j} = M_i \times M_{i+1} \times \dots \times M_j$$

- ▶ Existe $i \leq k < j$ tal que a última multiplicação realizada é:

$$M = M_{i,k} \times M_{k+1,j}$$

Com $b_{i-1} \cdot b_k \cdot b_j$ multiplicações.

- ▶ Se essa parentização em k for ótima, o número de multiplicações para computar $M_{i,k}$ e $M_{k+1,j}$ também deve ser ótimo.

Encontramos uma **subestrutura ótima**.

- ▶ Os produtos $M_{i,k}$ e $M_{k+1,n}$ devem ter parentizações ótimas.



Subproblema

Definimos o seguinte **subproblema**

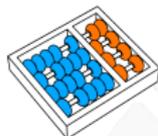
- ▶ Considere uma tabela m com entrada para cada par (i, j) .
- ▶ Seja $m[i, j]$ o valor de uma parentização ótima para:

$$M_i \times M_{i+1} \times \dots \times M_j$$

Podemos resolver esse subproblema recursivamente:

- ▶ Não são feitas multiplicações se $i = j$, então $m[i, j] = 0$.
- ▶ Do contrário, deve haver uma última multiplicação.
- ▶ Listando as possibilidades, obtemos uma **recorrência**:

$$m[i, j] = \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + b_{i-1} \cdot b_k \cdot b_j\}$$



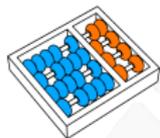
Algoritmo recursivo

Algoritmo: MÍNIMO-MULTIPLICAÇÕES-RECURSIVO(b, i, j)

```

1 se  $i = j$ 
2    $m[i, j] \leftarrow 0$ 
3 senão
4    $m[i, j] \leftarrow \infty$ 
5   para  $k \leftarrow i$  até  $j - 1$ 
6      $q \leftarrow$  MÍNIMO-MULTIPLICAÇÕES-RECURSIVO( $b, i, k$ )
7       + MÍNIMO-MULTIPLICAÇÕES-RECURSIVO( $b, k + 1, j$ )
8       +  $b[i - 1] \cdot b[k] \cdot b[j]$ 
9     se  $m[i, j] > q$ 
10       $m[i, j] \leftarrow q$ 
11       $s[i, j] \leftarrow k$ 
12 devolva  $m[i, j]$ 
    
```

- ▶ $s[i, j]$ guarda o índice para a última multiplicação de $M_{i,j}$.
- ▶ A chamada inicial é MÍNIMO-MULTIPLICAÇÕES-RECURSIVO($b, 1, n$).



Recuperando a solução ótima

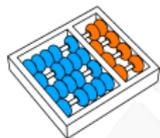
- ▶ A função anterior devolve apenas o valor da solução.
- ▶ Mas a tabela s induz uma **parentização ótima**.

Algoritmo: MULTIPLICA-MATRIZES(M, s, i, j)

```

1 se  $i = j$ 
2   imprima " $M_i$ "
3 senão
4   imprima "("
5   MULTIPLICA-MATRIZES( $M, s, i, s[i, j]$ )
6   MULTIPLICA-MATRIZES( $M, s, s[i, j] + 1, j$ )
7   imprima ")"

```



Complexidade do algoritmo recursivo

Seja $n = j - i + 1$ o número de matrizes.

- ▶ O tempo de execução é dado pela recorrência:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ \sum_{k=1}^{n-1} [T(k) + T(n-k)] + \Theta(n) & \text{se } n > 1 \end{cases}$$

- ▶ Podemos mostrar que $T(n) = \Omega(2^n)$.
- ▶ Isso ainda é **IMPRATICÁVEL**.



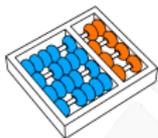
Complexidade do algoritmo recursivo

Analisando com calma:

- ▶ O algoritmo **recalcula** o mesmo valor $m[i, j]$ várias vezes.
- ▶ Por exemplo, se $n = 4$, os valores $m[1, 2]$, $m[2, 3]$ e $m[3, 4]$ são computados duas vezes.

Melhorando:

- ▶ O número de pares (i, j) distintos é limitado por $O(n^2)$.
- ▶ Podemos guardar todos os valores $m[i, j]$ em uma tabela.
- ▶ Resolvendo cada subproblema apenas uma vez.



Memorização x programação dinâmica

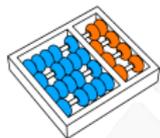
Evitamos o recálculo de subproblemas de duas maneiras.

1. Memorização:

- ▶ Mantemos a estrutura recursiva do algoritmo.
- ▶ Guardamos os valores computados em tabela.
- ▶ Devolvemos valores já conhecidos antes da chamada recursiva.

2. Programação dinâmica:

- ▶ Criamos uma tabela com entrada para cada subproblema.
- ▶ Inicializamos as entradas correspondentes a casos básicos.
- ▶ Preenchemos o restante da tabela com uma **recorrência**.
- ▶ A ordem de preenchimento deve obedecer a dependência entre subproblemas.



Algoritmo de memorização

Algoritmo: MÍNIMO-MULTIPLICAÇÕES-MEMORIZADO(b, i, j)

```

1 se  $m[i, j]$  não está definido
2   se  $i = j$ 
3      $m[i, j] \leftarrow 0$ 
4   senão
5      $m[i, j] \leftarrow \infty$ 
6     para  $k \leftarrow i$  até  $j - 1$ 
7        $q \leftarrow$  MÍNIMO-MULTIPLICAÇÕES-MEMORIZADO( $b, i, k$ )
8         + MÍNIMO-MULTIPLICAÇÕES-MEMORIZADO( $b, k + 1, j$ )
9         +  $b[i - 1] \cdot b[k] \cdot b[j]$ 
10      se  $m[i, j] > q$ 
11         $m[i, j] \leftarrow q$ 
12         $s[i, j] \leftarrow k$ 
13 devolva  $m[i, j]$ 
    
```



Algoritmo de programação dinâmica

- ▶ Denotamos por u o tamanho da cadeia do subproblema.
- ▶ Preenchemos a tabela em ordem crescente de u .

Algoritmo: MÍNIMO-MULTIPLICAÇÕES(b)

```

1  para  $i \leftarrow 1$  até  $n$ 
2  |    $m[i, i] \leftarrow 0$ 
3  para  $u \leftarrow 2$  até  $n$ 
4  |   para  $i \leftarrow 1$  até  $n - u + 1$ 
5  |   |    $j \leftarrow i + u - 1$ 
6  |   |    $m[i, j] \leftarrow \infty$ 
7  |   |   para  $k \leftarrow i$  até  $j - 1$ 
8  |   |   |    $q \leftarrow m[i, k] + m[k + 1, j] + b[i - 1] \cdot b[k] \cdot b[j]$ 
9  |   |   |   se  $q < m[i, j]$ 
10  |   |   |   |    $m[i, j] \leftarrow q$ 
11  |   |   |   |    $s[i, j] \leftarrow k$ 
12 devolva  $m[1, n]$ 

```



Complexidade da programação dinâmica

Análise:

- ▶ O algoritmo preenche cada entrada (i, j) uma vez.
- ▶ O número de pares diferentes na tabela é $O(n^2)$.
- ▶ Preencher cada entrada leva tempo $O(n)$.
- ▶ O tempo total é limitado número de pares \times tempo por par.
- ▶ Além disso, usamos uma matriz $n \times n$ para a tabela.

Complexidade:

- ▶ O tempo gasto pelo algoritmo é $O(n^3)$.
- ▶ A memória usada pelo algoritmo é $O(n^2)$.

PROGRAMAÇÃO DINÂMICA

MO417 - Complexidade de
Algoritmos I

Santiago Valdés Ravelo
[https://ic.unicamp.br/~santiago/
ravelo@unicamp.br](https://ic.unicamp.br/~santiago/ravelo@unicamp.br)

04/24

13



UNICAMP

