

## Capítulo 11

# Programação Dinâmica

A metodologia conhecida como programação dinâmica foi inventada pelo matemático americano Richard Bellman por volta de 1950 como um método genérico para otimizar processos de decisão. Assim, a palavra programação está mais relacionada com a ideia de planejamento, e não com programação de computadores. Depois de se estabelecer como uma importante técnica em Matemática Aplicada, a programação dinâmica passou a ser utilizada como uma estratégia de dividir para conquistar juntamente com uma tabela [4, 5], pois ao invés de resolver os subproblemas recursivamente, os mesmos são resolvidos sequencialmente e as soluções são armazenadas em uma tabela. Desta forma, esta metodologia é utilizada para resolver problemas subdividindo-os em subproblemas como na estratégia de dividir para conquistar, mas com uma diferença fundamental: os subproblemas se sobrepõem, e para evitar que o mesmo subproblema seja calculado mais de uma vez, os resultados são armazenados em uma tabela. Por exemplo, considere o problema de computar o  $n$ -ésimo número de Fibonacci através da seguinte recorrência:

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{se } n > 1 \end{cases} \quad (11.1)$$

Uma implementação direta da definição acima tem complexidade exponencial.

De fato, neste caso cada subproblema é calculado diversas vezes! A ideia para resolver este problema de forma mais eficiente é evitar refazer computações já feitas anteriormente:

---

**Algorithm 16:** fib( $n$ )

---

```
1 fib[i] = i, if i < 2;
2 for i = 2 to n do
3   | fib[i] = fib[i - 1] + fib[i - 2]
4 end
5 return fib[n]
```

---

No pseudocódigo acima, a linha 1 é executada em tempo constante, o **for** das linhas 2-4 é executado  $n$  vezes, de forma que o tempo de execução deste algoritmo é linear!

Os problemas que podem ser resolvidos usando programação dinâmica normalmente estão relacionados com *otimização*. No exemplo anterior, encontramos uma forma de *minimizar* o número de somas necessárias para calcular  $\text{fib}[n]$ . Para que esta metodologia possa ser aplicada é importante que o problema a ser resolvido satisfaça o *princípio da subestrutura ótima*: as soluções ótimas do problema contêm as soluções ótimas dos subproblemas. Assim, para desenvolver um algoritmo utilizando programação dinâmica, costumamos seguir os seguintes passos:

1. Caracterizar a estrutura de uma solução ótima;
2. Recursivamente definir o valor de uma solução ótima;
3. Computar o valor de uma solução ótima (usualmente de baixo para cima);
4. Construir uma solução ótima a partir da informação computada.

## 11.1 O problema do corte das hastes

Consideremos um outro problema de otimização: o problema do corte das hastes. Suponha que uma empresa deseja cortar hastes de forma a maximizar o valor total obtido pela venda dos pedaços cortados. Assumiremos os seguintes fatos:

1. O corte não tem custo;
2. As hastes são cortadas em pedaços cujos comprimentos são números inteiros.
3. O preço de uma haste de comprimento  $i \geq 1$  é igual a  $p_i$ .

O problema do corte das hastes pode, então, ser apresentado da seguinte forma: Dados uma haste de comprimento  $n$  e uma tabela com o preço  $p_i$  da haste de comprimento  $1 \leq i \leq n$ , determine a melhor forma de cortar a haste de comprimento  $n$  de forma a obter o valor máximo da venda dos pedaços resultantes do corte.

De quantas formas distintas podemos cortar uma haste de comprimento  $n$ ? Note que temos  $n - 1$  possíveis pontos de corte em uma haste de comprimento  $n$ .

Suponha que uma solução ótima divide a haste em  $1 \leq k \leq n$  pedaços. Assim,  $n = i_1 + i_2 + \dots + i_k$ , onde  $i_j$  denota o comprimento do  $j$ -ésimo pedaço da haste. O valor a ser obtido a partir da venda destes  $k$  pedaços é  $v(n) = p_{i_1} + p_{i_2} + \dots + p_{i_k}$ . Nosso objetivo é determinar  $k$  de forma que  $v(n)$  seja máximo. A recursão que corresponde ao valor máximo a ser obtido é dada por:

$$v(n) = \max\{p_n, v(1) + v(n-1), v(2) + v(n-2), \dots, v(n-1) + v(1)\}$$

Podemos simplificar esta recursão observando que a divisão da haste consiste em fazer o primeiro corte obtendo um pedaço de comprimento  $i$  que não será mais dividido, e um segundo pedaço de comprimento  $n-i$  que ainda será dividido de forma a maximizar o valor a ser obtido:

$$v(n) = \max_{1 \leq i \leq n} \{p_i + v(n - i)\} \quad (11.2)$$

O pseudocódigo a seguir implementa a computação correspondente à recursão (11.2).

---

**Algorithm 17:** corte-haste( $p, n$ )

---

```

1 if  $n = 0$  then
2   | return 0;
3 end
4  $q = -\infty$ ;
5 for  $i = 1$  to  $n$  do
6   |  $q = \max\{q, p[i] + \text{corte-haste}(p, n - i)\}$ ;
7 end
8 return  $q$ ;

```

---

O tempo  $T(n)$  de execução deste algoritmo é dado por

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

que tem solução exponencial. Isto não é surpreendente porque o algoritmo corte-haste considera todas as  $2^{n-1}$  possíveis formas de cortar uma haste de comprimento  $n$ .

Utilizando programação dinâmica, cada subproblema será resolvido apenas uma vez:

---

**Algorithm 18:** corte-hasteDP( $p, n$ )

---

```

1 let  $r[0..n]$  be a new array;
2  $r[0] = 0$ ;
3 for  $j = 1$  to  $n$  do
4   |  $q = -\infty$ ;
5   | for  $i = 1$  to  $j$  do
6     |  $q = \max\{q, p[i] + r[j - i]\}$ ;
7   | end
8   |  $r[j] = q$ ;
9 end
10 return  $r[n]$ ;

```

---

O tempo  $T(n)$  de execução deste algoritmo é determinado pelo número de vezes que o **for** das linhas 5-7 é executado:

$$T(n) = \sum_{j=1}^n \sum_{i=1}^j 1 = \sum_{j=1}^n j = \frac{n \cdot (n + 1)}{2}$$

e portanto o algoritmo é quadrático no tamanho da entrada.

## 11.2 Multiplicação de uma cadeia de matrizes

Consideremos agora um outro problema: Multiplicação de uma cadeia de matrizes. Queremos computar o produto  $A_1 \cdot A_2 \cdot \dots \cdot A_n$  de forma a executar o

menor número possível de multiplicações. Utilizaremos o algoritmo padrão para multiplicação de duas matrizes:

---

**Algorithm 19:** mult-matrix( $A, B$ )

---

```

1 if  $A.columns \neq B.rows$  then
2   | error “incompatible dimensions”
3 end
4 else
5   | let  $C$  be a new  $A.rows \times B.columns$  matrix;
6   for  $i = 1$  to  $A.rows$  do
7     | for  $j = 1$  to  $B.columns$  do
8       |    $c_{ij} = 0$ ;
9       |   for  $k = 1$  to  $A.columns$  do
10        | |  $c_{ij} = c_{ij} + a_{ik}.b_{kj}$ 
11        | | end
12        | | end
13      | end
14      | return  $C$ 
15 end

```

---

O número exato de multiplicações  $T(n)$  realizadas pelo algoritmo acima corresponde ao número de vezes que a linha 10 é executada: Suponha que as dimensões das matrizes  $A$  e  $B$  sejam, respectivamente iguais a  $p \times q$  e  $q \times r$ , ou seja, a matriz  $A$  possui  $p$  linhas e  $q$  colunas, enquanto que a matriz  $B$  possui  $q$  linhas e  $r$  colunas. Então o total de multiplicações é dado por:

$$\sum_{i=1}^p \sum_{j=1}^r \sum_{k=1}^q 1 = \sum_{i=1}^p \sum_{j=1}^r q = \sum_{i=1}^p (r \cdot q) = p \cdot q \cdot r$$

Em particular, se  $n = p = q = r$  então  $T(n) = n^3$ .

Assim, para multiplicarmos duas matrizes de dimensões respectivamente iguais a  $10 \times 4$  e  $4 \times 20$ , realizaremos  $10 \cdot 4 \cdot 20 = 800$  multiplicações. Suponha que estejamos interessados em multiplicar as matrizes  $A$ ,  $B$  e  $C$  (nesta ordem) de dimensões respectivamente iguais a  $10 \times 4$ ,  $4 \times 20$  e  $20 \times 32$ . Dado que o produto de matrizes é associativo, sabemos que o produto  $A.B.C$  pode ser realizado de duas formas distintas, a saber:  $(A.B).C$  ou  $A.(B.C)$ . O número de multiplicações necessárias para computar o produto  $(A.B).C$  utilizando o algoritmo acima é igual ao número de multiplicações para computar o produto  $A.B$ , que já sabemos ser igual a 800, mais o número de multiplicações necessárias para computar o produto de  $A.B$  com  $C$ , que é igual a  $10 \cdot 20 \cdot 32 = 6400$ ; ou seja, no total precisamos de  $800 + 6400 = 7200$  multiplicações para computar o produto  $(A.B).C$ . Para computar o produto  $A.(B.C)$  precisamos de  $4 \cdot 20 \cdot 32 = 2560$  multiplicações para computar  $B.C$ , mais  $10 \cdot 4 \cdot 32 = 1280$  multiplicações para computar o produto de  $A$  com  $B.C$  perfazendo um total de  $2560 + 1280 = 3840$  multiplicações. Portanto, é mais “econômico” multiplicar  $A.(B.C)$  do que  $(A.B).C$ . Este exemplo, nos leva a um problema mais geral que tentaremos resolver:

---

Suponha que queiramos multiplicar  $n$  matrizes  $A_1, A_2, \dots, A_n$  ( $n \geq 1$ ). Isto é, queremos computar o produto  $A_1.A_2.\dots.A_n$ . Como fazer isto de forma a realizar o menor número de multiplicações possível? Em outras palavras, como devemos associar o produto  $A_1.A_2.\dots.A_n$  de forma a minimizar o número de multiplicações a serem realizadas?

---

Para motivarmos a utilização de programação dinâmica para resolver este problema, vejamos que a abordagem ingênua (força-bruta) não é eficiente. A abordagem ingênua consiste em escolher a melhor dentre todas as associações possíveis para o produto  $A_1, A_2, \dots, A_n$ . Seja  $P(n)$  o número total de distintas formas de associar o produto em consideração. Quando  $n = 1$ , temos apenas uma matriz, e portanto  $P(1) = 1$  (produto degenerado). Para  $n > 1$ ,  $P(n) = \sum_{k=1}^{n-1} P(k).P(n-k)$ . É possível mostrar que  $P(n) = \Omega(2^n)$ , e portanto a solução ingênua é exponencial.

Antes de construirmos uma solução usando programação dinâmica, vejamos que este problema satisfaz o princípio da subestrutura ótima. Denote por  $A_{i..j}$  o resultado do produto  $A_i.A_{i+1}.\dots.A_j$ , onde  $i \leq j$ . Agora considere que uma associação ótima para  $A_i.A_{i+1}.\dots.A_j$  separa o produto entre  $A_k$  e  $A_{k+1}$ , *i.e.* a associação ótima será dada pela associação ótima de  $A_{i..k}$  e  $A_{k+1..j}$ .

**Questão:** A solução ótima de associar o produto  $A_i.\dots.A_k$  na solução de  $A_i.A_{i+1}.\dots.A_j$  é também uma solução ótima para  $A_i.\dots.A_k$ ? Sim, pois uma possível solução mais eficiente de associar  $A_i.\dots.A_k$  nos permitiria substituir a solução eficiente de  $A_i.A_{i+1}.\dots.A_j$  produzindo uma solução melhor do que a ótima, o que é uma contradição.

Se denotarmos por  $m[i, j]$  o número mínimo de multiplicações necessárias para multiplicar as matrizes  $A_i.A_{i+1}.\dots.A_j$  ( $i \leq j$ ), então podemos caracterizar o problema de determinar o número mínimo de multiplicações para computar o produto  $A_1.A_2.\dots.A_n$  através da recursão

$$m[1, n] = \min_{1 \leq k \leq n} \{m[1, k] + m[k+1, n] + p_0.p_k.p_n\} \quad (11.3)$$

assumindo que a matriz  $A_i$  ( $1 \leq i \leq n$ ) tem dimensão  $p_{i-1} \times p_i$ .

Em geral,  $m[i, j]$ , com  $1 \leq i \leq j \leq n$ , é dada por

$$m[i, j] = \begin{cases} 0, & \text{se } i = j \\ \min_{i \leq k \leq j} \{m[i, k] + m[k+1, j] + p_{i-1}.p_k.p_j\}, & \text{se } i < j \end{cases} \quad (11.4)$$

A matriz  $m[i, j]$  fornece o custo das soluções ótimas dos subproblemas, mas não nos fornece toda a informação necessária para a construção da solução ótima. Para isto, definiremos a matriz  $s[i, j]$  como sendo o valor  $k$  que indica onde o produto  $A_i.A_{i+1}.\dots.A_j$  deve ser dividido, ou seja,  $s[i, j]$  é o valor  $k$  tal que  $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}.p_k.p_j$ .

O algoritmo a seguir computa as matrizes  $m$  e  $s$ , enquanto que o algoritmo print-optimal-parens imprime o produto  $A_i \cdot A_j$  com a parentização correspondente à associação ótima:

---

**Algorithm 20:** matrix-chain-order( $p$ )

---

```
1  $n = p.length - 1;$ 
2 let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables;
3 for  $i = 1$  to  $n$  do
4    $m[i, i] = 0;$ 
5 end
6 for  $l = 2$  to  $n$  do
7   for  $i = 1$  to  $n - l + 1$  do
8      $j = i + l - 1;$ 
9      $m[i, j] = \infty;$ 
10    for  $k = i$  to  $j - 1$  do
11       $q = m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j;$ 
12      if  $q < m[i, j]$  then
13         $m[i, j] = q;$ 
14         $s[i, j] = k;$ 
15      end
16    end
17  end
18 end
```

---

**Algorithm 21:** print-optimal-parens( $s, i, j$ )

---

```
1 if  $i == j$  then
2   print  $A_i;$ 
3 end
4 else
5   print "(";
6   print-optimal-parens( $s, i, s[i, j]$ );
7   print-optimal-parens( $s, s[i, j] + 1, j$ );
8   print ")";
9 end
```

---

Assim, matrix-chain-order tem complexidade de tempo  $\Theta(n^3)$ , e complexidade de espaço  $\Theta(n^2)$ .