

# Técnicas de Projeto (Parte 4)

## Projeto e Análise de Algoritmos

Felipe Cunha

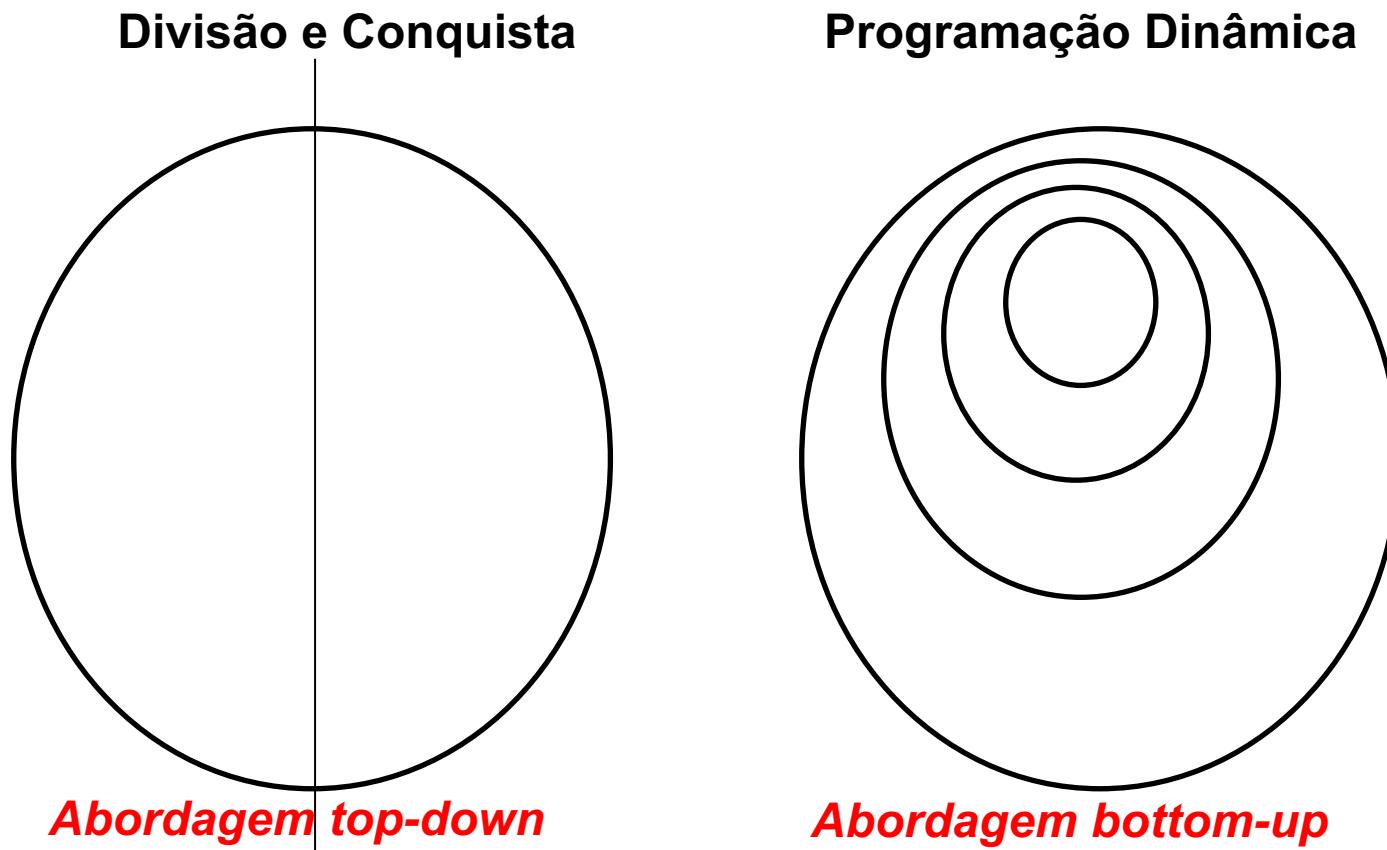
Pontifícia Universidade Católica de Minas Gerais

# Programação Dinâmica

- Programação não está relacionado com um programa de computador.
  - A palavra está relacionada com um método de solução baseado em tabela.
- Programação dinâmica (PD) × Divisão-e-conquista (DC):
  - DC quebra o problema em sub-problemas menores.
  - PD resolve todos os sub-problemas menores mas somente reusa as soluções ótimas.

# Programação Dinâmica

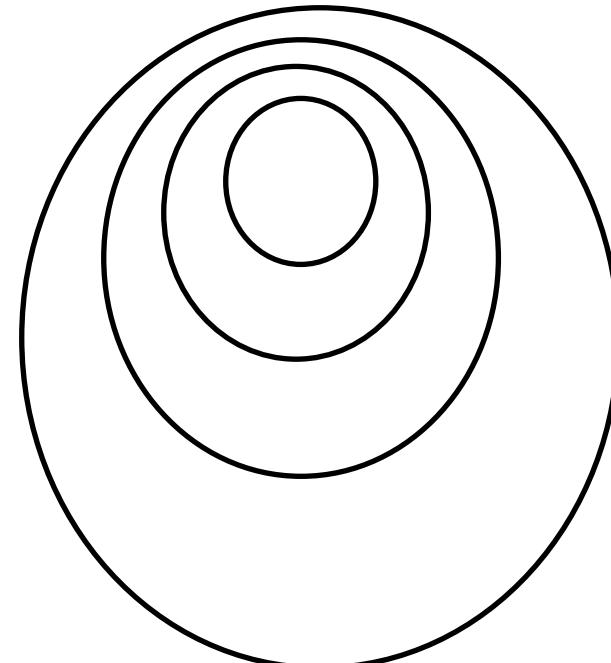
- Programação dinâmica quebra o problema em vários problemas sobrepostos.



# Programação Dinâmica

- Programação dinâmica quebra o problema em vários problemas sobrepostos.
- É um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória.
- É aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada (evitar recálculo) de outros subproblemas que, sobrepostos, compõem o problema original.

## Programação Dinâmica



# Programação Dinâmica

- O que um problema de otimização deve ter para que a **programação dinâmica** seja aplicável são duas principais características:
  1. Subestrutura ótima:
    - Quando uma solução ótima para o problema contém em seu interior soluções ótimas para subproblemas.
  2. Superposição de subproblemas:
    - Quando um algoritmo recursivo reexamina o mesmo problema muitas vezes.

# Programação Dinâmica

- Quando  $\sum \text{Tamanhos dos sub-problemas} = O(n)$ 
  - É provável que o algoritmo recursivo tenha **complexidade polinomial**.
- Quando a divisão de um problema de tamanho n resulta em:
  - $n$  Sub-problemas  $\times$  Tamanho  $n - 1$  cada um
  - É provável que o algoritmo recursivo tenha **complexidade exponencial**.
- Nesse caso, a técnica de programação dinâmica pode levar a um algoritmo mais eficiente.
  - A programação dinâmica calcula a solução para todos os sub-problemas, partindo dos sub-problemas menores para os maiores, armazenando os resultados em uma tabela.
  - A vantagem é que uma vez que um sub-problema é resolvido, a resposta é armazenada em uma tabela e nunca mais é recalculado.

# Sequência de Fibonacci

- Suponha o exemplo da sequencia de Fibonacci, implementada de forma recursiva
  - Quando a forma recursiva é implementada sem maiores cuidados, sem memorização, o seu cálculo de tempo cresce exponencialmente.

```
Se n <= 1 então F(n) := 1
caso contrário F(n) := F(n-1) + F(n-2)
```

- Quando implementada de forma recursiva "ingênua" (naive), sem memorização, a dupla chamada à F na segunda linha, causa a necessidade da repetição de cálculos, que cresce exponencialmente.

# Sequência de Fibonacci

- Suponha o exemplo da sequencia de Fibonacci, implementada de forma recursiva
  - Observe que a rigor esse caso não é um problema de programação dinâmica!
    - O cálculo do  $n$ -ésimo número de Fibonacci cresce linearmente, e não exponencialmente.
  - Porém este exemplo ainda assim é utilizado, pois a simplicidade é grande.

# Sequência de Fibonacci

- Fibonacci com PD:
  - Solução iterativa: criar um vetor para armazenar os valores anteriores.

1	1	2	3	5	8	...
---	---	---	---	---	---	-----

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

# Sequência de Fibonacci

- Fibonacci com PD:

- Solução iterativa: criar um vetor para armazenar os valores anteriores.

```
// PD, bottom up, não usa recursividade e armazena os resultados anteriores em Array

int DP_Bottom_Up(int n) {
    memo[0] = memo[1] = 1; // Valores padrões
    //de 2 a n ( ja sabemos que fib(0) e fib(1) = 1)
    for (int i=2; i<=n; i++) {
        memo[i] = memo[i-1] + memo[i-2];
    }
    return memo[n];
}
```

- Tabela de memo(n) guarda valor dos somatórios; para cada entrada basta acrescentar 1 termo

# Problema do Troco das Moedas

- Dado um conjunto de **n moedas**, cada uma com valor  $c_i$ , e, dada uma quantidade  $P$ , precisamos achar o **numero mínimo de moedas** para obter aquela quantidade
- Definimos a função  $D(i, Q)$  como o número mínimo de moedas necessário para obter uma quantidade  $Q$ , usando os  $i$  primeiros tipos de moedas ( $1 \dots i$ ).
- A solução da função Troco pode utilizar o ou mais moedas do tipo  $i$ 
  - Se não usa nenhuma moeda do tipo  $i$  então:
    - $D(i, Q) = D(i-1, Q)$
  - Se usa sim,  $k$  moedas do tipo  $i$ , então:
    - $D(i, Q) = D(i-1, Q - k \cdot c_i) + k$

# Problema do Troco das Moedas

- $M = \{1, 2, 4, 5, 8\}$
- $S = 17$

<i>m</i>	<i>n</i>																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9
4	0	1	1	2	1	2	2	3	2	3	3	4	3	4	4	5	4	5
5	0	1	1	2	1	1	2	2	2	2	2	3	3	3	3	3	4	4
8	0	1	1	2	1	1	2	2	1	2	2	3	2	2	3	3	2	3

# Problema do Troco das Moedas

- $M = \{1, 2, 6, 8\}$
- $S = 12$

	$n$												
	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	1	1	2	2	3	3	4	4	5	5	6	6
6	0	1	1	2	2	3	1	2	2	3	3	4	2
8	0	1	1	2	2	3	1	2	1	2	2	3	2

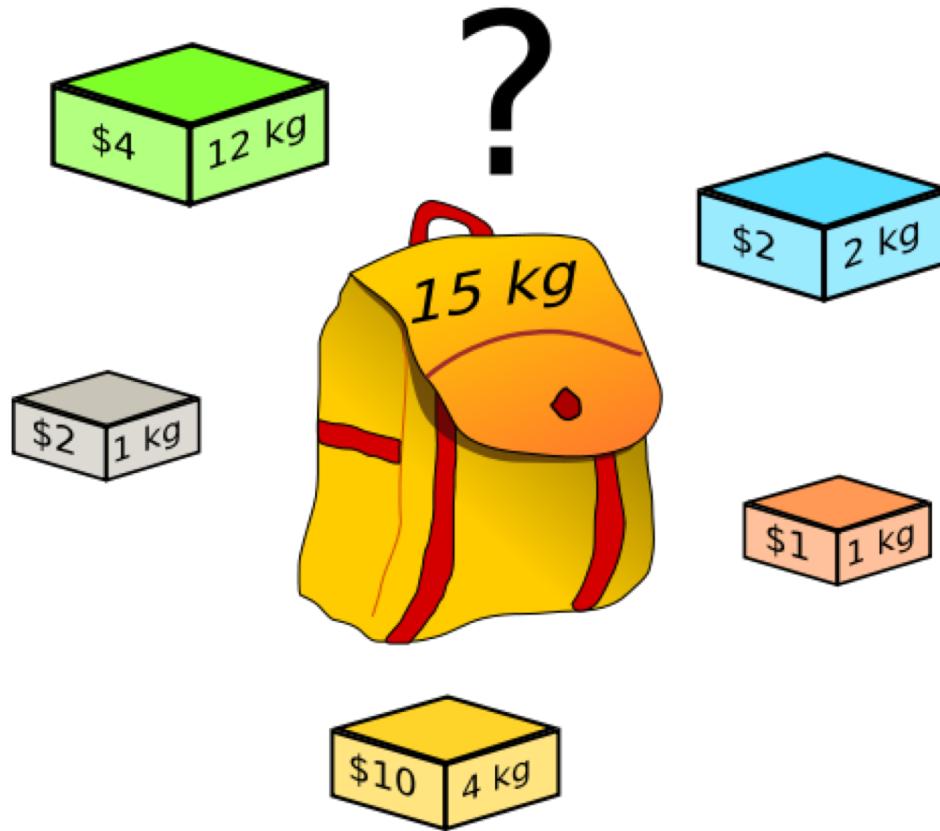
# Problema do Troco das Moedas

- $M=\{1,4,6\}$
- $S= 8$

# Problema do Troco das Moedas

```
Moedas ( int P, int N, int C[ ], int D[ ][ ]){  
    int i,j;  
    For ( i=0;i <=N; i++)  
        D[i][0]=0;  
    For ( i=1;i <=P; i++)  
        D[0][i]=NUM_MAX;  
    For (i=1; i<=N; i++){  
        For(j=1;j<=P;j++){  
            If (C[i] > j )  
                D[i][j] = D[ i-1][j];  
            Else if (D[i-1][j] < (D[i][j-C[i]]+1))  
                D[i][j]=D[i-1][j];  
            Else  
                D[i][j]= D[i][j-C[i]]+1;  
        }  
    }  
}
```

# Problema da Mochila



# Problema da Mochila

- Este é um problema de otimização e como tal possui muitas soluções possíveis, onde cada uma delas tem um valor
- Desejamos descobrir uma solução com o valor ótimo de acordo com um critério de minimização ou maximização.
- Existem muitas variações deste problema, consideraremos a mais simples delas:
  - Definição Mochila( $n, K$ ): dados um inteiro  $K$  e  $n$  itens de tamanhos diferentes tal que o  $i$ -ésimo item tem um tamanho inteiro  $k_i$ , descobrir um subconjunto de itens cujos tamanhos somam exatamente  $K$ , ou determine que nenhum tal subconjunto existe.

# Problema da Mochila

- Observações
  - A mochila pode ser um caminhão, navio, chip de silício, etc.
  - $\text{Mochila}(i,j)$  denotará o problema com os primeiros  $i$  itens e uma mochila de tamanho  $j$ .
  - Por simplicidade, nos concentraremos apenas em descobrir se uma solução existe.

# Problema da Mochila

- Algoritmo Mochila ( $S, K$ )
- Entrada: um vetor  $S$  de tamanho  $n$  armazenando os tamanhos dos itens e o tamanho  $K$  da mochila.
- Saída: um vetor bidimensional  $P$  tal que
  - $P(i,j).existe = \text{true}$  se existe uma solução com os primeiros  $i$  elementos para uma mochila de tamanho  $j$ ;
  - $P(i,j).pertence = \text{true}$  se o  $i$ -ésimo elemento pertence a solução.

# Problema da Mochila

## Início

```
P(0,0).existe = true;  
para j = 1 até K faça  
    P(0,j).existe = false;  
    para i = 1 até n faça  
        para j = 0 até K faça  
            P(i,j).existe = false;  
            if P(i - 1, j).existe então  
                P(i,j).existe = true;  
                P(i,j).pertence = false;  
            senão se (j - S(i) >= 0) então  
                se P(i - 1, j - S(i)).existe então  
                    P(i,j).existe = true;  
                    P(i,j).pertence = true.
```

## Fim

# Problema da Mochila

- O problema da mochila (em inglês, Knapsack problem) é um problema de otimização combinatória
- Nome devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores
- O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo

# Problema da Mochila

$$T(i, j) = \begin{cases} T(i-1, j) & \text{se } R[i] > S[j] \\ \max(T(i-1, j), T(i-1, S[j]-R[i]) + V[i]) & \text{se } R[i] \leq S[j] \end{cases}$$

# Problema da Mochila

**Capacidade 10**

<i>Peso</i>	<i>Lucro</i>	<i>Itens</i>	0	1	2	3	4	5	6	7	8	9	10
w1=3	x1=5	1	0	0	0	5	5	5	10	10	10	15	15
w2=4	x2=7	2	0	0	0	5	7	7	10	12	14	15	17
w3=5	x3=8	3	0	0	0	5	7	8	10	12	14	15	17

# Problema da Mochila

**Capacidade 10**

<i>Peso</i>	<i>Lucro</i>	<i>Itens</i>	0	1	2	3	4	5	6	7	8	9	10
w1=3	x1=5	1	0	0	0	5	5	5	10	10	10	15	15
w2=4	x2=7	2	0	0	0	5	7	7	10	12	14	15	17
w3=5	x3=8	3	0	0	0	5	7	8	10	12	14	15	17

# Multiplicação de Matrizes

- Seja

$$\mathbf{M} = \mathbf{M}_1 \times \mathbf{M}_2 \times \dots \times \mathbf{M}_n$$

- onde  $\mathbf{M}_i$  é uma matriz com  $d_{i-1}$  linhas e  $d_i$  colunas,  $2 \leq i \leq n$ .
- Isto serve para dizer apenas que a matriz  $\mathbf{M}_i$  possui uma quantidade de linhas igual a quantidade de colunas de  $\mathbf{M}_{i-1}$  ( $d_{i-1}$ ) e uma quantidade de colunas dada por  $d_i$ .
- A ordem da multiplicação pode ter um efeito enorme no número total de operações de adição e multiplicação necessárias para obter  $\mathbf{M}$ .
- Considere o produto de uma matriz  $p \times q$  por outra matriz  $q \times r$  cujo algoritmo requer  $O(pqr)$  operações.

# Multiplicação de Matrizes

- Considere o produto

$$\mathbf{M} = \mathbf{M}_1[10, 20] \times \mathbf{M}_2[20, 50] \times \mathbf{M}_3[50 \times 1] \times \mathbf{M}_4[1, 100]$$

- onde as dimensões de cada matriz aparecem entre colchetes.
- Sejam duas possíveis ordens de avaliação dessa multiplicação:

$$M = M_1 \times (M_2 \times (\underbrace{M_3 \times M_4}))$$

$$50 \times 1 \times 100 = 5\,000 \text{ operações}$$

$$M = (M_1 \times (\underbrace{M_2 \times M_3})) \times M_4$$

$$20 \times 50 \times 1 = 1\,000 \text{ operações}$$

$$M = M_1 \times (\underbrace{M_2 \times M_a}), \text{ sendo } M_a[50, 100]$$

$$20 \times 50 \times 100 = 100\,000 \text{ operações}$$

$$M = (\underbrace{M_1 \times M_a}) \times M_4, \text{ sendo } M_a[20, 1]$$

$$10 \times 20 \times 1 = 200 \text{ operações}$$

$$M = \underbrace{M_1 \times M_b}, \text{ sendo } M_b[20, 100]$$

$$10 \times 20 \times 100 = 20\,000 \text{ operações}$$

$$M = \underbrace{M_b \times M_4}, \text{ sendo } M_b[10, 1]$$

$$10 \times 1 \times 100 = 1\,000 \text{ operações}$$

$$\text{Total} = 125\,000 \text{ operações}$$

$$\text{Total} = 2\,200 \text{ operações}$$

# Multiplicação de Matrizes

- Tentar todas as ordens possíveis para minimizar o número de operações  $f(n)$  é exponencial em  $n$ , onde  $f(n) \geq 2^{n-2}$ .
- Usando programação dinâmica é possível obter um algoritmo  $O(n^3)$ .
- Seja  $m_{ij}$  o menor custo para computar  $M_i \times M_{i+1} \times \dots \times M_j$ , para  $1 \leq i \leq j \leq n$ .

# Multiplicação de Matrizes

$$m_{ij} = \begin{cases} 0, & \text{se } i = j, \\ \min_{i \leq k < j} (m_{ik} + m_{k+1,j} + d_{i-1}d_kd_j), & \text{se } j > i. \end{cases}$$

- $m_{ik}$  representa o custo mínimo para calcular  $\mathbf{M}' = \mathbf{M}_i \times \mathbf{M}_{i+1} \times \dots \times \mathbf{M}_k$ .
- $m_{k+1,j}$  representa o custo mínimo para calcular  $\mathbf{M}'' = \mathbf{M}_{k+1} \times \mathbf{M}_{k+2} \times \mathbf{M}_j$ .
- $d_{i-1}d_kd_j$  representa o custo de multiplicar  $\mathbf{M}' [d_{i-1}; d_k]$  por  $\mathbf{M}'' [d_k; d_j]$ .
- $m_{ij}; j > i$  representa o custo mínimo de todos os valores possíveis de  $k$  entre  $i$  e  $j - 1$ , da soma dos três termos.

# Multiplicação de Matrizes

- A solução usando programação dinâmica calcula os valores de  $m_{ij}$  na ordem crescente das diferenças nos subscritos.
- O cálculo inicia com  $m_{ii}$  para todo  $i$ , depois  $m_{i,i+1}$  para todo  $i$ , depois  $m_{i,i+2}$ , e assim sucessivamente.
- Desta forma, os valores  $m_{ik}$  e  $m_{k+1,j}$  estarão disponíveis no momento de calcular  $m_{ij}$ .
- Isto acontece porque  $j - i$  tem que ser estritamente maior do que ambos os valores de  $k - i$  e  $j - (k + 1)$  se  $k$  estiver no intervalo  $i \leq k < j$ .

# Multiplicação de Matrizes

**AVALIAMULTMATRIZES( $n, d[0..n]$ )**

▷ Parâmetro:  $n$  (nº de matrizes);  $d[0..n]$  (dimensões das matrizes)

▷ Constante e variáveis auxiliares:

- $MaxInt = \text{maior inteiro}$
- $i, j, k, h, n, temp$
- $m[1..n, 1..n]$

```

1  for  $i \leftarrow 1$  to  $n$ 
2    do  $m[i, i] \leftarrow 0$ 
3  for  $h \leftarrow 1$  to  $n - 1$ 
4    do for  $i \leftarrow 1$  to  $n - h$ 
5      do  $j \leftarrow i + h$ 
6         $m[i, j] \leftarrow MaxInt$ 
7        for  $k \leftarrow i$  to  $j - 1$  do
8           $temp \leftarrow m[i, k] + m[k + 1, j] + d[i - 1] \times d[k] \times d[j]$ 
9          if  $temp < m[i, j]$ 
10         then  $m[i, j] \leftarrow temp$ 
11  print  $m$ 

```

- A execução de AVALIAMULTMATRIZES obtém o custo mínimo para multiplicar as  $n$  matrizes, assumindo que são necessárias  $pqr$  operações para multiplicar uma matriz  $p \times q$  por outra matriz  $q \times r$ .

# Multiplicação de Matrizes

A multiplicação de

$$M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100],$$

sendo

$d$	10	20	50	1	100
	0	1	2	3	4

produz como resultado

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10.000$	$m_{23} = 1.000$	$m_{34} = 5.000$	
$M_1 \times M_2$	$M_2 \times M_3$	$M_3 \times M_4$	
$m_{13} = 1.200$	$m_{24} = 3.000$		
$M_1 \times (M_2 \times M_3)$	$(M_2 \times M_3) \times M_4$		
$m_{14} = 2.200$			
$(M_1 \times (M_2 \times M_3)) \times M_4$			

# Problema Subsequência Comum Mais Longa

- Dada uma sequência  $X = \langle x_1, x_2, \dots, x_m \rangle$ , outra sequência  $Z = \langle z_1, z_2, \dots, z_k \rangle$  é uma subsequência de  $X$  se existe uma sequência estritamente crescente  $\langle i_1, i_2, \dots, i_k \rangle$  de índices de  $X$  tais que, para todo  $j = 1, 2, \dots, k$ , temos:

$$X_i = Z_j$$

- Exemplo:
- $X = \langle A, B, C, B, D, A, B \rangle$  e  $Z = \langle B, C, D, B \rangle$   $Z$  é uma subsequência de  $X$  com sequência de índice correspondente  $\langle 2, 3, 5, 7 \rangle$ .

# Problema Subsequência Comum Mais Longa

- Dada duas sequências X e Y, dizemos que uma sequência Z é uma **subsequência comum** de X e Y se Z é uma subsequência de X e de Y ao mesmo tempo.
- Exemplo:
- $X = \langle A, B, C, B, D, A, B \rangle$  e  $Y = \langle B, D, C, A, B, A \rangle$  a sequência  $\langle B, C, A \rangle$  é uma subsequência comum. A sequência  $\langle B, C, B, A \rangle$  é uma subsequência comum mais longa (LCS – longest common subsequence)

# Problema Subsequência Comum Mais Longa

$$c[i, j] = \begin{cases} 0 & \text{se } i, j > 0 \quad x_i = y_i \\ c[i - 1, j - 1] + 1 & \text{se } i, j > 0 \quad x_i \neq y_i \\ \max(c[i, j - 1], c[i - 1, j]) & \text{se } i, j > 0 \quad x_i \neq y_i \end{cases}$$

# Problema Subsequência Comum Mais Longa

	yj	A	C	T	G	T	G	C	A
xi	0	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3	3
T	0	1	2	3	3	4	4	4	4
G	0	1	2	3	4	4	5	5	5
T	0	1	2	3	4	5	5	5	5
C	0	1	2	3	4	5	5	6	6
A	0	1	2	3	4	5	5	6	7

# Problema Subsequência Comum Mais Longa

	$y_j$	A	C	T	G	T	G	C	A
$x_i$	0	0	0	0	0	0	0	0	0
A	0	↖	↖	↖	↖	↖	↖	↖	↖
C	0	↑	↖	↖	↖	↖	↖	↖	↖
G	0	↑	↑	↑	↖	↖	↖	↖	↖
T	0	↑	↑	↖	↑	↖	↖	↖	↖
G	0	↑	↑	↑	↖	↑	↖	↖	↖
T	0	↑	↑	↖	↑	↖	↑	↑	↑
C	0	↑	↖	↑	↑	↑	↑	↖	↖
A	0	↖	↑	↑	↑	↑	↑	↑	↖

Maior subsequência comum: ACGTGCA

# Problema Subsequência Comum Mais Longa

```
LCS-LENGTH(X, Y)
1 m ← comprimento [X]
2 n ← comprimento [Y]
3 for i ← 0 to m
4   do c[i, 0] ← 0
5 for j ← 0 to n
6   do c[0, j] ← 0
7 for i ← 1 to m
8   do for j ← 0 to n
9     do if xi = yj
10       then c[i, j] ← c[i-1, j-1] + 1
11       b[i, j] ← " "
12     else if c[i-1, j] ≥ c[i, j-1]
13       then c[i, j] ← c[i-1, j]
14       b[i, j] ← "↑"
15     else c[i, j] ← c[i, j-1]
16       b[i, j] ← "←"
17 return c e b
```

# Problema Subsequência Comum Mais Longa

```
PRINT-LCS(b, x, i, j)
1 if i = 0 ou j = 0
2   then return
3   if b[i, j] = " "
4     then PRINT-LCS(b, x, i, j)
5     print xi
6   else if b[i, j] = "↑"
7     then PRINT-LCS(b, x, i-1, j)
8   else PRINT-LCS(b, x, i-1, j)
```

# Programação Dinâmica: Princípio de Optimalidade

- A solução eficiente está baseada no **princípio da optimalidade**:
  - Em uma seqüência ótima de escolhas ou de decisões cada sub-seqüência deve também ser ótima.
- Cada sub-seqüência representa o custo mínimo, assim como  $m_{ij}$ ,  $j > i$ .
- Assim, todos os valores da tabela representam escolhas ótimas.

# Programação Dinâmica: Princípio de Otimalidade

- O princípio da otimalidade não pode ser aplicado indiscriminadamente.
- Se o princípio não se aplica é provável que não se possa resolver o problema com sucesso por meio de programação dinâmica.
  - Quando, por exemplo, o problema utiliza recursos limitados e o total de recursos usados nas sub-instâncias é maior do que os recursos disponíveis.
- Exemplo do princípio da otimalidade: suponha que o caminho mais curto entre Belo Horizonte e Curitiba passa por Campinas. Logo,
  - O caminho entre BH e Campinas também é o mais curto possível;
  - Como também é o caminho entre Campinas e Curitiba;
- Logo, o princípio da otimalidade se aplica.

# Programação Dinâmica: Princípio de Optimalidade

- Seja o problema de encontrar o caminho mais longo entre duas cidades. Temos que:
  - Um caminho simples nunca visita uma mesma cidade duas vezes.
  - Se o caminho mais longo entre Belo Horizonte e Curitiba passa por Campinas, isso não significa que o caminho possa ser obtido tomando o caminho simples mais longo entre Belo Horizonte e Campinas e depois o caminho simples mais longo entre Campinas e Curitiba.
- Observe que o caminho simples mais longo entre BH e Campinas pode passar por Curitiba!
  - Quando os dois caminhos simples são agrupados não existe uma garantia que o caminho resultante também seja simples.
  - Logo, o princípio da optimalidade não se aplica.

# Programação Dinâmica

- **Quando aplicar PD?**
  - Respeitam as duas características principais.
  - Problema computacional deve ter uma formulação recursiva.
  - Não deve haver ciclos na formulação (usualmente o problema deve ser reduzido a problemas menores).
  - Número total de instâncias do problema a ser resolvido deve ser pequeno ( $n$ ).
  - Tempo de execução é  $O(n) \times$  tempo para resolver a recursão.

# Exercícios

1. Seja o problema da maior subsequência crescente, projete uma solução, para encontrar o tamanho da maior subsequência.
2. Seja uma estrada pedagiada. Os pedágios devem estar nas cidades ao longo da rodovia distantes pelo menos  $x$  Km. Cada pedágio colocado dará um lucro de  $y_i$ . Projete um algoritmo para instalar os pedágios que produzirá o maior lucro possível.