# Programação Dinâmica

Eduardo Camponogara

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina

DAS-9003: Introdução a Algoritmos

# Sumário

# Introdução

## Programação Dinâmica

- Uma técnica poderosa para resolver problemas de decisão sequencial e de controle consiste em quebrá-los em subproblemas menores, mais fáceis de serem resolvidos.
- Quando quebramos um problema em subproblemas do mesmo tipo, tipicamente podemos produzir algoritmos recursivos, frequentemente empregados para resolver problemas computacionais.

# Introdução

Dois paradigmas de concepção de algoritmos por meio de quebra:

**Divisão e Conquista:** este paradigma é amplamente adotado na concepção de algoritmos tipicamente encontrados na Ciência da Computação

**Programação Dinâmica:** este paradigma consiste em quebrar um problema em uma sequência de decisões que são tomadas em estágios.

# Introdução

## Divisão e Conquista

- Este paradigma é amplamente adotado na concepção de algoritmos tipicamente encontrados na Ciência da Computação
- É dividido em três etapas:
  - quebre o problema em duas metades;
  - resolva cada metade separadamente; e
  - combine as metades de forma a obter uma solução completa

# Introdução

## Divisão e Conquista

- O algoritmo de intercalação ou *merge-sort*, por exemplo, ordena uma cadeia de $n$ números em $\Theta(n \log n)$ operações de comparação segundo o princípio de divisão e conquista.
- O algoritmo "constrói" implicitamente uma árvore de subproblemas, tendo profundidade $\log_2 n$ e largura $n$.
- Uma vez que o custo computacional para solução dos subproblemas em cada nível é linear, o algoritmo ordena a cadeia em $\Theta(n \log n)$ operações elementares.

# Introdução

## Programação Dinâmica

- Este paradigma consiste em quebrar um problema em uma sequência de decisões que são tomadas em estágios.
- Em outras palavras, o paradigma segue os passos:
  - remova um elemento do problema;
  - resolva o problema menor; e
  - use a solução do problema menor para adicionar o elemento removido de maneira adequada, produzindo uma solução para o problema maior.

# Programação Dinâmica

- Este tipo de abordagem é comum em problemas de controle ótimo
- Às vezes é possível conceber uma solução analítica fechada para cada estágio o que resulta em um algoritmo eficiente
- Mesmo nos casos onde apenas uma solução iterativa pode ser obtida, a programação dinâmica pode resultar em uma solução algorítmica eficaz.

# Programação Dinâmica

## Observações

Aspectos importantes sobre PD (Programação Dinâmica) são:

1) problemas cujas instâncias apresentam uma ordem da esquerda para a direita, como cadeias de símbolos e vértices de polígonos, são candidatos a uma solução por PD;
2) sem uma ordem, PD pode resultar em um algoritmo de tempo exponencial; e
3) PD produz uma solução ótima global.

# Programação Dinâmica

## O Que Vamos Estudar?

- Estudaremos o formalismo de programação dinâmica de uma forma informal, através de exemplos.
- Começamos com o problema de calcular os números de Fibonacci, que ilustra a economia computacional que pode ser obtida por meio da programação dinâmica.
- O exemplo seguinte mostra como PD pode ser empregada na solução do problema da mochila, exemplificando uma maneira de se obter uma solução ótima a partir dos valores ótimos das funções objetivo.

# Programação Dinâmica

## O Que Vamos Estudar?

- O terceiro exemplo trata da identificação da subsequência de símbolos mais longa, enquanto que o último exemplo apresenta a solução para um problema complexo de edição de cadeias de símbolos.
- Este último problema e suas generalizações têm aplicações em sistemas operacionais, casamento de padrões e sequenciamento de DNA.

# Sumário

# Programação Dinâmica: Princípios

## Princípios

- Programação dinâmica é uma técnica difícil de entender mas cuja solução, uma vez obtida, é de fácil compreensão.
- Em algoritmos para problemas como o de ordenação de números, corretude é mais fácil de verificar do que eficiência.
- Este não é o caso de *problemas de otimização*, onde deseja-se provar que um algoritmo sempre produz uma solução ótima.

# Programação Dinâmica: Princípios

## Princípios

- Algoritmos gulosos, heurísticas que executam uma busca local são *eficientes* mas apenas *ocasionalmente* encontram uma solução ótima.
- Algoritmos baseados em enumeração, por outro lado, avaliam direta ou indiretamente todas as possíveis soluções e portanto encontram a solução ótima, todavia o custo computacional pode ser proibitivo.

# Programação Dinâmica: Princípios

## Princípios

- *Programação Dinâmica* combina estes dois universos (algoritmos gulosos e de enumeração):
  - A técnica sistematicamente considera todas as possíveis decisões e sempre seleciona a ótima
  - As consequências de todas as decisões até o presente estágio
  - Usando estas informações de uma forma sistemática, o trabalho computacional pode ser minimizado.

# Série de Fibonacci

A sequência de Fibonacci é dada por

$$\begin{aligned} F_k &= F_{k-1} + F_{k-2} \\ F_0 &= 0 \\ F_1 &= 1 \end{aligned}$$

- A série de Fibonacci foi inicialmente proposta e estudada por um matemático no contexto de reprodução animal.
- Ela apresenta aplicações em Teoria dos Números e Computação.

# Série de Fibonacci

## Algoritmo recursivo para calcular $F_k$

Algoritmo $F(k)$

```
if k = 0
   return 0
else
   if k = 1
      return 1
   else
      return F(k − 1) + F(k − 2)
```

# Série de Fibonacci

Qual é a complexidade do algoritmo?

# Série de Fibonacci

## Qual É a Complexidade do Algoritmo?

Para calcularmos o número de operações elementares, inicialmente obtemos uma recorrência:

$$\begin{cases} T(n) = 1 & \text{para } n = 0, 1 \\ T(n) = T(n-1) + T(n-2) & \text{para } n \geq 2 \end{cases}$$

# Série de Fibonacci

## Qual É a Complexidade do Algoritmo?

Limite inferior para $T(n)$:

$$\begin{aligned} T(n) &\geq T(n-2) + T(n-2) \\ &= 2T(n-2) \\ &\geq 4T(n-4) \\ &\geq 8T(n-6) \\ &\vdots \\ &\geq 2^k T(n-2k) \end{aligned}$$

# Série de Fibonacci

- Note que

$$n - 2k = 1 \Leftrightarrow k = \frac{n-1}{2}$$

portanto, $T(n) \geq 2^{\lfloor \frac{n-1}{2} \rfloor} \Rightarrow T(n) \in \Omega(2^{n/2})$.

- Concluímos então que o algoritmo recursivo é ineficiente, levando um tempo exponencial para calcular $F_k$.

# Série de Fibonacci

- Você pode mostrar que o algoritmo executa em tempo $\Theta(2^n)$?
- Veja a árvore de recursão dada na Figura 1.

# Série de Fibonacci

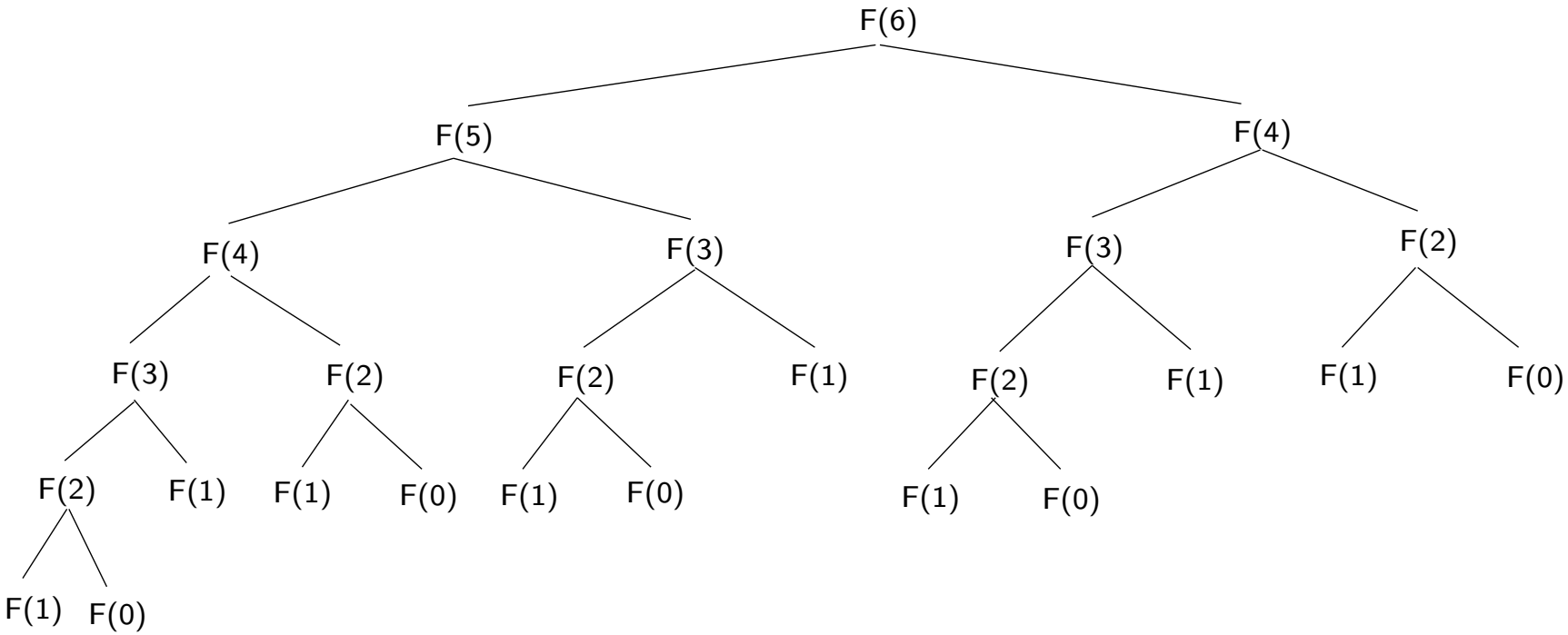



Figura: Árvore de recursão do algoritmo recursivo para cálculo do número de Fibonacci

# Série de Fibonacci

## Versão mais eficiente

Podemos desenvolver um algoritmo mais eficiente, de tempo linear se armazenarmos os valores das instâncias menores e não recalcularmos os seus valores.

# Série de Fibonacci

## Algoritmo de programação dinâmica

O algoritmo alternativo é dado abaixo.

`Algoritmo` $F_b(n)$

---

$F_0 = 0$
$F_1 = 1$
`for` $i \leftarrow 2$ `to` $n$
    $F_i \leftarrow F_{i-2} + F_{i-1}$
Return $F_n$

---

# Série de Fibonacci

## Aspectos do algoritmo

- Uma vez que calculamos os números de Fibonacci, começando com os menores e armazenando os resultados, teremos já calculado os valores de $F_{i-1}$ e $F_{i-2}$ quando computarmos o valor de $F_i$.
- O algoritmo é executado em tempo $\Theta(n)$.

# Sumário

# O Problema da Mochila

O problema da mochila é definido em programação matemática:

$$
\begin{aligned}
P : z = \quad & Max \quad \sum_{j=1}^{n} c_j x_j \\
& s.\ a : \quad \sum_{j=1}^{n} a_j x_j \leq b \\
& \qquad\quad x_j \in \{0,1\},\ j = 1, \ldots, n
\end{aligned}
$$

onde os coeficientes $\{a_j\}_{j=1}^{n}$ e $b$ são inteiros positivos.

# O Problema da Mochila

## Princípio para aplicação de programação dinâmica

Imagine que o lado direito da desigualdade assume um valor $\lambda$ que varia de 0 até $b$, o que define estágios do problema de programação dinâmica, cujas soluções ótimas são $x_0, \ldots, x_k, \ldots, x_b$.

# O Problema da Mochila

Isto nos leva a definir o problema $P_k(\lambda)$, a sua respectiva solução ótima $x_k(\lambda)$ e o seu respectivo valor objetivo $f_k(\lambda)$ como segue:

$$
\begin{aligned}
P_k(\lambda) : f_k(\lambda) = \quad & Max \quad \sum_{j=1}^{k} c_j x_j \\
& s.\ a : \quad \sum_{j=1}^{k} a_j x_j \leq \lambda \\
& \qquad\quad x_j \in \{0,1\},\ j = 1, \ldots, k
\end{aligned}
$$

# O Problema da Mochila

- Em palavras, $P_k(\lambda)$ é o problema da mochila restrito aos $k$ primeiros itens e uma mochila de capacidade $\lambda$.
- Assim, $z = f_n(b)$ nos dá a solução ótima para $P$.
- Nos resta então obter uma recursão que permite calcular $f_k(\lambda)$ em termos dos valores de $f_s(u)$ com $s \leq k$ e $u \leq \lambda$.

# Programação Dinâmica para o Problema da Mochila

O que podemos dizer sobre a solução ótima $x^*$ para o problema $P_k(\lambda)$ com valor $f_k(\lambda)$?

Claramente, $x_k^* = 0$ ou $x_k^* = 1$.

Considerando cada caso, temos:

Caso 1) Se $x_k^* = 0$, então concluímos que a solução ótima satisfaz $f_k(\lambda) = f_{k-1}(\lambda)$

Caso 2) Se $x_k^* = 1$, então concluímos que a solução ótima satisfaz $f_k(\lambda) = c_k + f_{k-1}(\lambda - a_k)$

# Programação Dinâmica para o Problema da Mochila

O que podemos dizer sobre a solução ótima $x^*$ para o problema $P_k(\lambda)$ com valor $f_k(\lambda)$?

- Combinando os casos (1) e (2), obtemos a recorrência abaixo:

$$f_k(\lambda) = Max\{f_{k-1}(\lambda),\ c_k + f_{k-1}(\lambda - a_k)\} \qquad (1)$$

- Definindo-se os valores iniciais como $f_0(\lambda) = 0$ para $\lambda \geq 0$, pode-se utilizar a recorrência (1) para calcular sucessivamente os valores de $f_1, f_2, \ldots, f_n$ para todos os valores inteiros de $\lambda \in \{0, \ldots, b\}$.

# Programação Dinâmica para o Problema da Mochila

## Como determinar a solução ótima?

- A questão que resta é como encontrar a solução ótima associada ao valor ótimo.
- Podemos manter um indicador $p_k(\lambda)$ que assume valor 0 se $f_k(\lambda) = f_{k-1}(\lambda)$, e valor 1 caso contrário.

# Programação Dinâmica para o Problema da Mochila

## Como determinar a solução ótima?

A solução pode então ser encontrada por meio dos passos abaixo:

- se $p_n(b) = 0$, então como $f_n(b) = f_{n-1}(b)$, definimos $x_n^* = 0$ e continuamos o processo com o valor $f_{n-1}(b)$;
- se $p_n(b) = 1$, então $f_n(b) = c_n + f_{n-1}(b - a_n)$, definimos $x_n^* = 1$ e repetimos este procedimento para $f_{n-1}(b - a_n)$;
- após $n$ iterações, obteremos a solução ótima.

# Programação Dinâmica para o Problema da Mochila

## Complexidade do Algoritmo de Programação Dinâmica

- Calculando o número de operações necessárias para obtermos $z = f_n(b)$, verificamos que o cálculo de $f_k(\lambda)$ para $\lambda = 0, 1, \ldots, b$ e $k = 1, \ldots, n$ necessita de um número constante de operações.
- O algoritmo roda em tempo $\Theta(nb)$, sendo portanto pseudo-polinomial.
- O algoritmo tem tempo de execução polinomial se $b \in O(\log n)$, já que $b$ pode ser representado em notação binária com $k$ bits, ou seja, $b \leq 2^k$.

# Problema da Mochila Exemplo

**Exemplo:** Considere a instância do problema da mochila que segue abaixo.

$$
\begin{array}{rcrrrrrrrrl}
z & = & \text{Maximize} & 10x_1 & + & 7x_2 & + & 25x_3 & + & 24x_4 & \\
 & & \text{sujeito a:} & 2x_1 & + & x_2 & + & 6x_3 & + & 5x_4 & \leq \quad 7
\end{array}
$$

# Problema da Mochila Exemplo

- Para se aplicar PD, calculamos os valores de $f_0(\lambda)$.
- A próxima coluna ($k = 1$) é calculada usando a fórmula (1).
- O resultado completo da aplicação do algoritmo de programação dinâmica está ilustrado nas tabelas abaixo.
- Nela são dados os valores da função objetivo de cada subproblema e as decisões relativas à inserção do item do estágio na mochila.

# Problema da Mochila Exemplo

| $\lambda$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | | | | | | | | |
| 1 | 0 | | | | | | | | |
| 2 | 0 | | | | | | | | |
| 3 | 0 | | | | | | | | |
| 4 | 0 | | | | | | | | |
| 5 | 0 | | | | | | | | |
| 6 | 0 | | | | | | | | |
| 7 | 0 | | | | | | | | |

# Problema da Mochila Exemplo

| $\lambda$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | | | | 0 | | | |
| 1 | 0 | 0 | | | | 0 | | | |
| 2 | 0 | 10 | | | | 1 | | | |
| 3 | 0 | 10 | | | | 1 | | | |
| 4 | 0 | 10 | | | | 1 | | | |
| 5 | 0 | 10 | | | | 1 | | | |
| 6 | 0 | 10 | | | | 1 | | | |
| 7 | 0 | 10 | | | | 1 | | | |

# Problema da Mochila Exemplo

| $\lambda$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | | 0 | 0 | | |
| 1 | 0 | 0 | 7 | | | 0 | 1 | | |
| 2 | 0 | 10 | 10 | | | 1 | 0 | | |
| 3 | 0 | 10 | 17 | | | 1 | 1 | | |
| 4 | 0 | 10 | 17 | | | 1 | 1 | | |
| 5 | 0 | 10 | 17 | | | 1 | 1 | | |
| 6 | 0 | 10 | 17 | | | 1 | 1 | | |
| 7 | 0 | 10 | 17 | | | 1 | 1 | | |

# Problema da Mochila Exemplo

| $\lambda$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | |
| 1 | 0 | 0 | 7 | 7 | | 0 | 1 | 0 | |
| 2 | 0 | 10 | 10 | 10 | | 1 | 0 | 0 | |
| 3 | 0 | 10 | 17 | 17 | | 1 | 1 | 0 | |
| 4 | 0 | 10 | 17 | 17 | | 1 | 1 | 0 | |
| 5 | 0 | 10 | 17 | 17 | | 1 | 1 | 0 | |
| 6 | 0 | 10 | 17 | 25 | | 1 | 1 | 1 | |
| 7 | 0 | 10 | 17 | 32 | | 1 | 1 | 1 | |

# Problema da Mochila Exemplo

| $\lambda$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 7 | 7 | 7 | 0 | 1 | 0 | 0 |
| 2 | 0 | 10 | 10 | 10 | 10 | 1 | 0 | 0 | 0 |
| 3 | 0 | 10 | 17 | 17 | 17 | 1 | 1 | 0 | 0 |
| 4 | 0 | 10 | 17 | 17 | 17 | 1 | 1 | 0 | 0 |
| 5 | 0 | 10 | 17 | 17 | 24 | 1 | 1 | 0 | 1 |
| 6 | 0 | 10 | 17 | 25 | 31 | 1 | 1 | 1 | 1 |
| 7 | 0 | 10 | 17 | 32 | 34 | 1 | 1 | 1 | 1 |

# Problema da Mochila Exemplo

| $\lambda$ | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 7 | 7 | 7 | 0 | 1 | 0 | 0 |
| 2 | 0 | 10 | 10 | 10 | 10 | **1** | **0** | **0** | 0 |
| 3 | 0 | 10 | 17 | 17 | 17 | 1 | 1 | 0 | 0 |
| 4 | 0 | 10 | 17 | 17 | 17 | 1 | 1 | 0 | 0 |
| 5 | 0 | 10 | 17 | 17 | 24 | 1 | 1 | 0 | 1 |
| 6 | 0 | 10 | 17 | 25 | 31 | 1 | 1 | 1 | 1 |
| 7 | 0 | 10 | 17 | 32 | 34 | 1 | 1 | 1 | **1** |

# Problema da Mochila Exemplo

Relembrando que $f_k(\lambda) = Max\{f_{k-1}(\lambda),\ c_k + f_{k-1}(\lambda - a_k)\}$, podemos reconstruir a solução ótima:

$$p_4(7) = 1 \Rightarrow x_4^* = 1$$
$$p_3(7-5) = p_3(2) = 0 \Rightarrow x_3^* = 0$$
$$p_2(2) = 0 \Rightarrow x_2^* = 0$$
$$p_1(2) = 1 \Rightarrow x_1^* = 1$$

A solução ótima é $x^*(1,0,0,1)$ e $f(x^*) = 34$.

# Sumário

# Elementos de um Algoritmo de Programação Dinâmica

Aqui ilustramos os elementos básicos de um algoritmo de programação dinâmica. Essencialmente, um algoritmo PD pode ser projetado em três passos:

1) Formule uma resposta como uma relação de recorrência
2) Mostre que o número de valores distintos de cada recorrência é limitada por um polinômio (de preferência de baixa ordem)
3) Especifique uma ordem para avaliar a recorrência de forma que sempre se faça cálculos sobre valores já conhecidos.

# Problema

## Problema: Sequência Crescente Mais Longa

- Dada uma sequência de $n$ números, e.g. $(9, 5, 2, 8, 7, 3, 1, 6, 4)$, deseja-se encontrar a "subsequência" mais longa cujos números estão em ordem crescente.
- No problema em consideração, a sequência mais longa para $s = (9, 5, 2, 8, 7, 3, 1, 6, 4)$ é de comprimento 3, podendo ser:
  - a subsequência $s_1 = (2, 3, 4)$ ou
  - a subsequência $s_2 = (2, 3, 6)$.

# Problema

- Se estivéssemos procurando uma sequência contígua, o problema seria facilmente resolvido.
- **Qual seria o número de soluções candidatas (contíguas)?** O número de soluções candidatas pode ser calculado pela recorrência:

$$\begin{aligned} T(n) &= n + (n-1) + (n-2) + \ldots + 1 \\ &= \frac{n(n+1)}{2} \Rightarrow T(n) \in \Theta(n^2) \end{aligned}$$

# Problema

Para a situação sob consideração, qual é o número de possíveis soluções candidatas?

$$
\begin{aligned}
T(n) &= \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n} \\
&= \sum_{k=0}^{n} \binom{n}{k} \\
&= (1+1)^n \qquad \text{(pelo Binômio de Newton)} \\
&= 2^n \Rightarrow T(n) \in \Theta(2^n)
\end{aligned}
$$

# Algoritmo de Programação Dinâmica

- Não é difícil construir um algoritmo com tempo de execução $\Theta(n^2)$ para encontrar a subsequência mais longa de uma cadeia de $n$ símbolos.
- Tomando como base o número de possíveis soluções para a versão do problema onde a subcadeia não é necessariamente contígua, $\Theta(2^n)$, nos parece totalmente inviável enumerar todas as possíveis soluções.
- Veremos a seguir que, apesar do número elevado de soluções candidatas, podemos projetar um algoritmo PD de tempo $\Theta(n^2)$.

# Algoritmo de Programação Dinâmica

- Suponha que para a sequência $s' = (s_1, s_2, \ldots, s_{n-1})$ conhecemos o comprimento $l_k$, $k \in \{1, \ldots, n-1\}$, da sequência mais longa que termina no elemento $s_k$ de $s'$.
- **Como podemos encontrar a sequência mais longa em $(s_1, \ldots, s_n)$?**

# Algoritmo de Programação Dinâmica

- Suponha que para a sequência $s' = (s_1, s_2, \ldots, s_{n-1})$ conhecemos o comprimento $l_k$, $k \in \{1, \ldots, n-1\}$, da sequência mais longa que termina no elemento $s_k$ de $s'$.
- **Como podemos encontrar a sequência mais longa em $(s_1, \ldots, s_n)$?**
- Isto pode ser realizado por meio da recorrência:

$$l_n = Max\{l_k + 1 : s_k \leq s_n,\ k = 1, \ldots, n-1\}$$

para a qual um algoritmo é dado a seguir.

# Algoritmo de Programação Dinâmica

Algoritmo $LS(s)$

---

$l_0 \leftarrow 0$
$s_0 \leftarrow 0$
$p_0 \leftarrow 0$ // predecessor, $p_i$ é o índice do elemento
que aparece imediatamente antes de $s_i$
na sequência que termina em $s_i$.
for $k = 1$ to $n$ do
$l_k = Max_{j=0,\ldots,k-1}\{l_j + 1 : s_j < s_k\}$
$p_k = ArgMax_{j=0,\ldots,k-1}\{l_j + 1 : s_j < s_k\}$
return $Max\{l_k : k = 1, \ldots, n\}$

---

# Algoritmo de Programação Dinâmica

- O funcionamento do algoritmo durante o cálculo da sequência mais longa da cadeia $s = (9, 5, 2, 8, 7, 3, 1, 6, 4)$ é ilustrado nas tabelas abaixo.
- O comprimento da cadeia mais longa é, portanto, $3 = Max\{l_i : i = 1, \ldots, n\}$ tendo como solução a subcadeia que termina no elemento 4 ou a subcadeia que termina no elemento 6.

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | | | | | | | | | |
| Comprimento $l_i$ | 0 | | | | | | | | | |
| Predecessor $p_i$ | 0 | | | | | | | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | | | | | | | | |
| Comprimento $l_i$ | 0 | 1 | | | | | | | | |
| Predecessor $p_i$ | 0 | 0 | | | | | | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | | | | | | | |
| Comprimento $l_i$ | 0 | 1 | 1 | | | | | | | |
| Predecessor $p_i$ | 0 | 0 | 0 | | | | | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | | | | | | |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | | | | | | |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | | | | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | 8 | | | | | |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | 2 | | | | | |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | 3 | | | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | 8 | 7 | | | | |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | 2 | 2 | | | | |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | 3 | 3 | | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | 8 | 7 | 3 | | | |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | 2 | 2 | 2 | | | |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | 3 | 3 | 3 | | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | 8 | 7 | 3 | 1 | | |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | | |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | 8 | 7 | 3 | 1 | 6 | |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 3 | |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | 6 | |

# Algoritmo de Programação Dinâmica

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Seqüência $s_i$ | 0 | 9 | 5 | 2 | 8 | 7 | 3 | 1 | 6 | 4 |
| Comprimento $l_i$ | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 3 | 3 |
| Predecessor $p_i$ | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | 6 | 6 |

# Algoritmo de Programação Dinâmica

## Qual é o tempo de execução do nosso algoritmo?

- Podemos determinar o tempo de execução através da recorrência $T(n) = T(n-1) + n$, o que nos leva a concluir que $T(n) \in \Theta(n^2)$.
- Utilizando estruturas de dados como dicionários de uma forma inteligente é possível encontrar a "sequência" mais longa em tempo $\Theta(n \log n)$.

# Sumário

# *Approximate String Matching*

## *String Matching*

- Uma tarefa importante em processamento de texto é a busca por ocorrências de uma palavra no texto.
- Infelizmente, palavras não são escritas corretamente, surgindo portanto a questão: **Como poderíamos efetuar uma busca pela cadeia mais próxima de um padrão fornecido pelo usuário?**
- Mais formalmente, seja $P$ uma cadeia padrão e $T$, uma cadeia que representa o texto.

# *Approximate String Matching*

- A "distância" entre $P$ e $T$ é o menor número de operações de edição necessárias para transformar $T$ em $P$, sendo as operações permitidas:
  a) **Substituição:** dois símbolos podem diferir: KAT $\rightarrow$ CAT
  b) **Inserção:** adiciona-se um símbolo de $P$ que não aparece em $T$: CT $\rightarrow$ CAT
  c) **Deleção:** elimina-se um caracter de $T$ que não aparece em $P$: CAAT $\rightarrow$ CAT

# *Approximate String Matching*

## Exemplo

$P = abcdefghijkl$ pode ser mapeado para $T = bcdeffghixkl$ usando três operações:

- inserção do caractere $a$ antes de $b$,
- deleção de um caractere $f$, e
- substituição do caractere $x$ pelo caractere $j$.

# Projeto de um Algoritmo DP

- Que informação seria necessária para tomarmos a decisão final?
- Isto é, o que deve acontecer na última posição de $T$?
- Possibilidades:
  a) o último caractere pode ser "emparelhado" com o último caractere de $P$, se eles são idênticos
  b) o último caractere pode ser substituído
  c) as outras operações são de deleção e inserção

# Construindo um Algoritmo

## Fundamentos

Seja $D[i, j]$ a "distância", isto é, o menor número de operações para emparelhamento entre $(P_1, P_2, \ldots, P_i)$ e $(T_1, T_2, \ldots, T_j)$.

# Construindo um Algoritmo

## Fundamentos

Então, $D[i,j]$ é o mínimo das três possíveis formas de estender as sub-cadeias.

1) Se $(P_i = T_j)$, então $D[i,j] \leftarrow D[i-1,j-1]$ (*emparelhamento*). Caso contrário, $D[i,j] \leftarrow D[i-1,j-1]+1$ (*substituição*).
2) $D[i,j] \leftarrow D[i-1,j]+1$, o que significa que há um caractere adicional na cadeia de busca $P$, portanto pagamos o custo de uma inserção e não avançamos o ponteiro do texto $T$.
3) $D[i,j] \leftarrow D[i,j-1]+1$, o que significa que há um caractere extra na cadeia $T$, portanto pagamos o custo de uma deleção em $T$ e não avançamos o ponteiro de $P$.

# Construindo um Algoritmo

## Elementos do algoritmo de programação dinâmica

- Entretanto, ainda falta especificar as condições iniciais.
- A inicialização correta dos valores é crítica para que o algoritmo retorne o resultado correto.
- Por exemplo, o valor $D[0, j]$ corresponde a emparelhar os $j$ primeiros símbolos do texto com nenhum caractere do padrão $P$.

# Construindo um Algoritmo

## Elementos do algoritmo de programação dinâmica

- Duas situações:
  a) Se desejamos um emparelhamento entre todo o padrão $P$ e todo o texto $T$, então $D[0, j]$ é o custo de eliminarmos $j$ símbolos de $T$. Assim, $D[0, j] = j$ para $j = 1, \ldots, |T|$.
  b) Se desejamos "emparelhar" $P$ com qualquer subcadeia de $T$, então o custo de iniciarmos o emparelhamento na posição $j$ do texto deve ser o mesmo de iniciarmos na posição 1. Assim, $D[0, j] = 0$ para $j = 1, \ldots, |T|$.

  Em ambos os casos, $D[i, 0] = i$ pois não podemos eliminar símbolos de $P$ sem pagarmos o preço de deleção.

# Construindo um Algoritmo

## Algoritmo de programação dinâmica

Antes de apresentarmos o algoritmo, assumimos que $m = |T|$ e $n = |P|$.

Algoritmo $EditDistance(P, T)$

---

for $i = 0$ to $n$ do $D[i, 0] = i$
for $j = 0$ to $m$ do $D[0, j] = j$

for$i = 1$ to $n$ do
    for$j = 1$ to $m$ do
    $D[i, j] = Min(D[i - 1, j - 1]$
      $+matchcost(P_i, T_j), D[i - 1, j] + 1, D[i, j - 1] + 1)$
Fim algoritmo

---

# Construindo um Algoritmo

## Qual é a complexidade do algoritmo?

A partir do pseudo-código do algoritmo *EditDistance* fica claro que o tempo de execução é da ordem $\Theta(mn)$.

# Exemplo

A tabela abaixo traz o resultado da operação $EditDistance(P, T)$ para $P = abcdefghijkl$ e $T = bcdeftghixkl$.

# Exemplo

| | T | b | c | d | e | f | f | g | h | i | x | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | 1 | | | | | | | | | | | | |
| b | 2 | | | | | | | | | | | | |
| c | 3 | | | | | | | | | | | | |
| d | 4 | | | | | | | | | | | | |
| e | 5 | | | | | | | | | | | | |
| f | 6 | | | | | | | | | | | | |
| g | 7 | | | | | | | | | | | | |
| h | 8 | | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | | |
| j | 10 | | | | | | | | | | | | |
| k | 11 | | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | | |

# Exemplo

| | T | b | c | d | e | f | f | g | h | i | x | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | 1 | 1[2] | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| b | 2 | | | | | | | | | | | | |
| c | 3 | | | | | | | | | | | | |
| d | 4 | | | | | | | | | | | | |
| e | 5 | | | | | | | | | | | | |
| f | 6 | | | | | | | | | | | | |
| g | 7 | | | | | | | | | | | | |
| h | 8 | | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | | |
| j | 10 | | | | | | | | | | | | |
| k | 11 | | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | | |

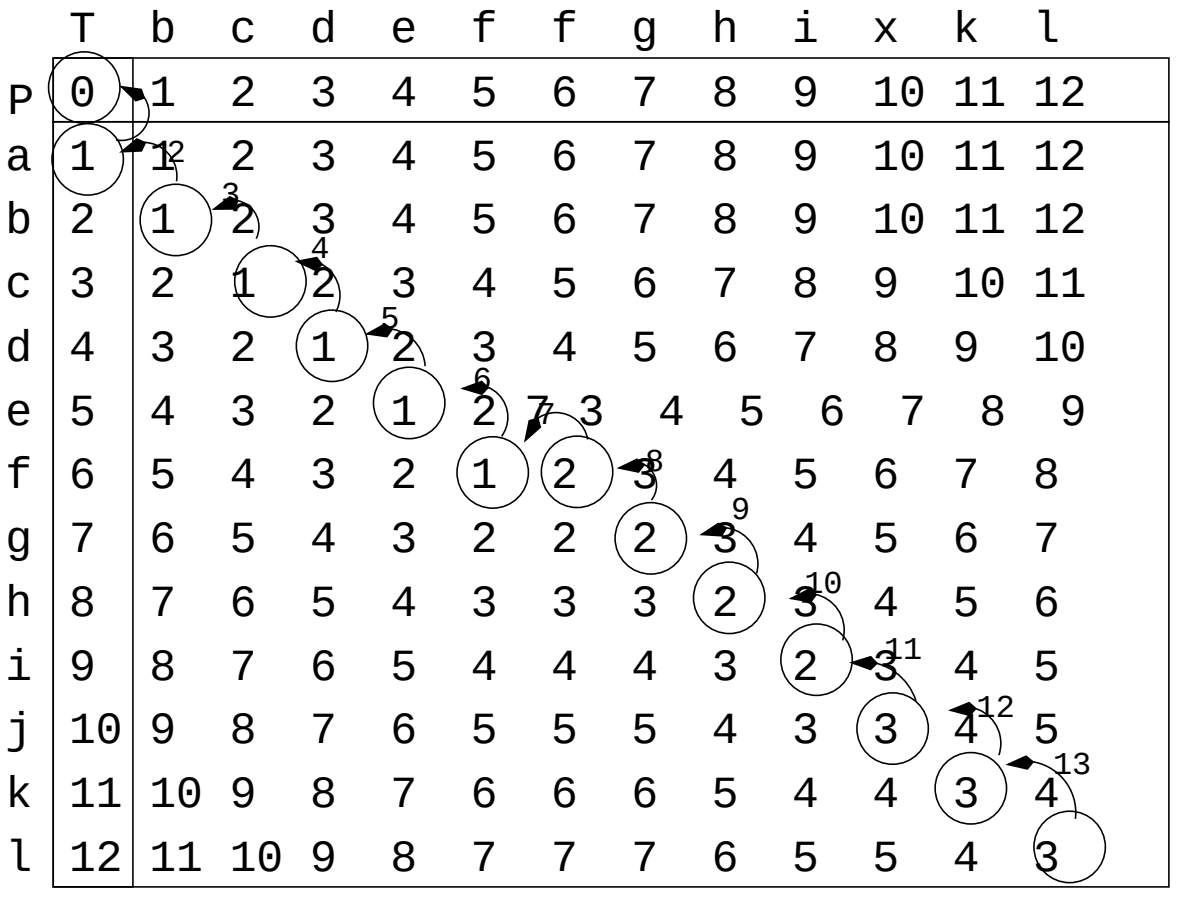
# Exemplo

| | T | b | c | d | e | f | f | g | h | i | x | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| b | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| c | 3 | | | | | | | | | | | | |
| d | 4 | | | | | | | | | | | | |
| e | 5 | | | | | | | | | | | | |
| f | 6 | | | | | | | | | | | | |
| g | 7 | | | | | | | | | | | | |
| h | 8 | | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | | |
| j | 10 | | | | | | | | | | | | |
| k | 11 | | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | | |

# Exemplo

| | T | b | c | d | e | f | f | g | h | i | x | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| b | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| c | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| d | 4 | | | | | | | | | | | | |
| e | 5 | | | | | | | | | | | | |
| f | 6 | | | | | | | | | | | | |
| g | 7 | | | | | | | | | | | | |
| h | 8 | | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | | |
| j | 10 | | | | | | | | | | | | |
| k | 11 | | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | | |

# Exemplo


T b c d e f f g h i x k l
P 0 1 2 3 4 5 6 7 8 9 10 11 12
a 1 1 2 3 4 5 6 7 8 9 10 11 12
b 2 1 2 3 4 5 6 7 8 9 10 11 12
c 3 2 1 2 3 4 5 6 7 8 9 10 11
d 4 3 2 1 2 3 4 5 6 7 8 9 10
e 5 4 3 2 1 2 3 4 5 6 7 8 9
f 6 5 4 3 2 1 2 3 4 5 6 7 8
g 7 6 5 4 3 2 2 2 3 4 5 6 7
h 8 7 6 5 4 3 3 3 2 3 4 5 6
i 9 8 7 6 5 4 4 4 3 2 3 4 5
j 10 9 8 7 6 5 5 5 4 3 3 4 5
k 11 10 9 8 7 6 6 6 5 4 4 3 4
l 12 11 10 9 8 7 7 7 6 5 5 4 3

# Exemplo

A Figura 33 traz em forma tabular o valores $D[i, j]$, $i = 0, \ldots, n$ e $j = 1, \ldots, m$, calculados pelo algoritmo de programação dinâmica.

# Exemplo

Abaixo listamos as operações executadas pelo algoritmo, conforme números 1 até 13 indicadas na figura.

1) insert (custo 1)
2) match (custo 0)
3) match (custo 0)
4) match (custo 0)
5) match (custo 0)
6) match (custo 0)
7) delete (custo 1)
8) match (custo 0)
9) match (custo 0)
10) match (custo 0)
11) substitute $x \rightarrow j$ (custo 1)
12) match (custo 0)
13) match (custo 0)

# Programação Dinâmica: Conclusões

- Fim!
- Obrigado pela presença