# Algoritmos em *Strings*

J. Pascoal Faria, R. Rossetti, L. Ferreira

FEUP, MIEIC, CAL, 2017/2018

# Índice

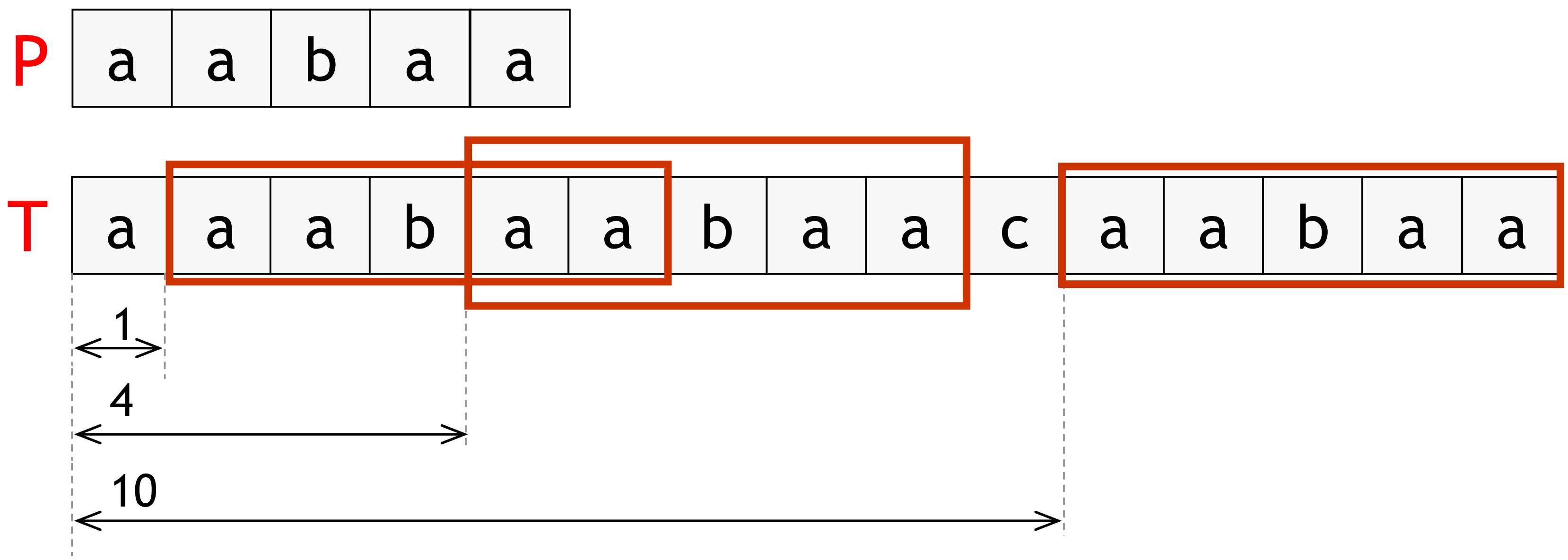
- Pesquisa exata (*string matching*)
- Pesquisa aproximada (*approximate string matching*)
- Outros problemas

# Pesquisa exata

# (*string matching*)

# Problema

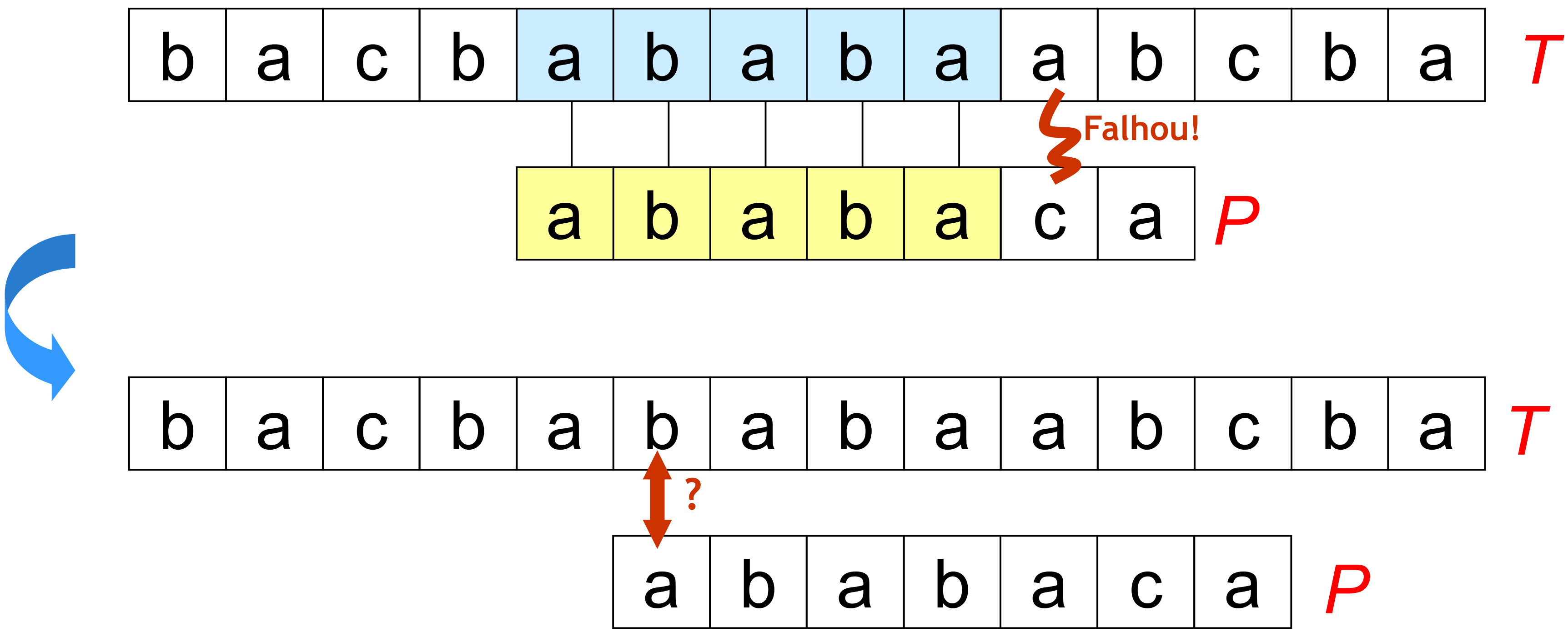
- Encontrar todas as ocorrências de um padrão P num texto T
  - P e T são cadeias de caracteres
  - Ocorrências são definidas pela deslocação em relação ao início do texto
  - Ocorrências podem ser sobrepostas



# Algoritmos

- Algoritmo *naive*
  - Para cada deslocamento possível, compara desde o início do padrão
  - Ineficiente se o padrão for comprido: O(|P|.|T|)
- Algoritmo baseado em autómato finito
  - Pré-processamento: gerar autómato finito correspondente ao padrão
  - Permite depois analisar o texto em tempo linear O(|T|), pois cada carácter só precisa de ser processado uma vez
  - Mas tempo e espaço requerido pelo pré-processamento pode ser elevado: O(|P|.|Σ|), em que |Σ| é o tamanho do alfabeto
- Algoritmo de Knuth-Morris-Pratt
  - Efetua um pré-processamento do padrão em tempo O(|P|), sem chegar a gerar explicitamente um autómato, seguido de processamento do texto em O(|T|), dando total O(|T|+|P|)

# Algoritmo *naive*

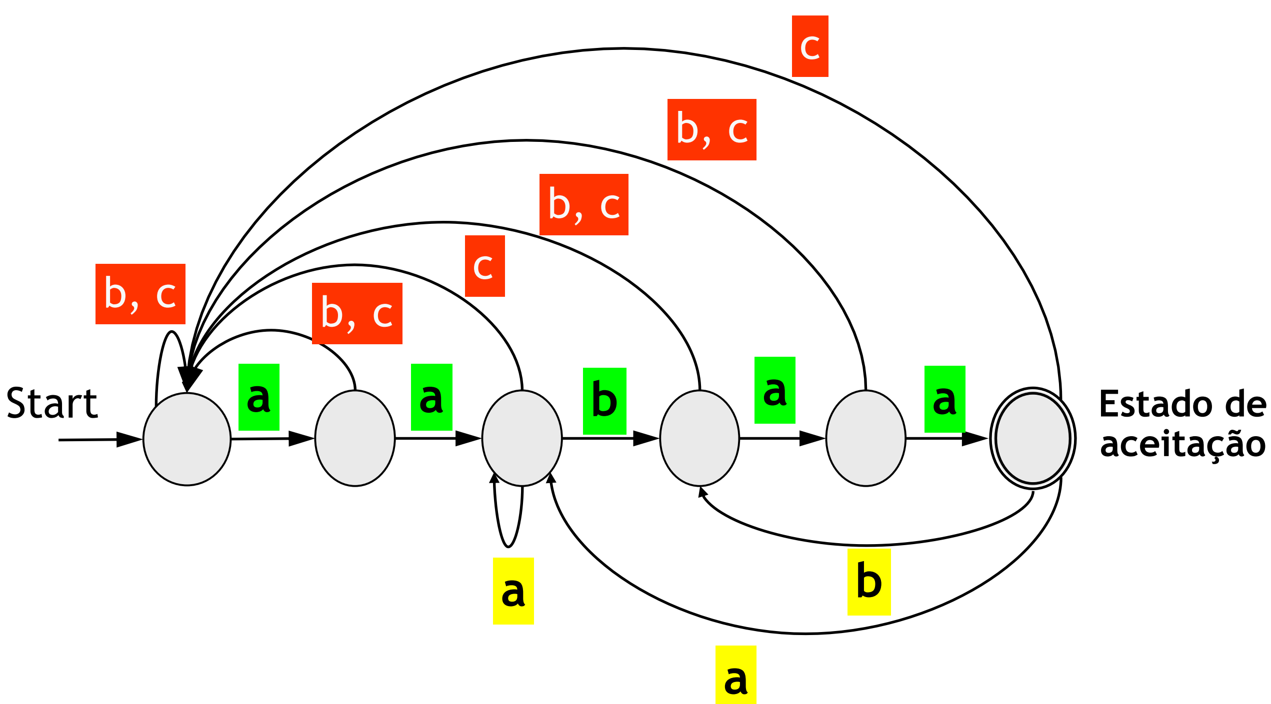



Desloca-se o padrão uma casa para a direita e recomeça-se a comparação do início do padrão!        Ineficiente: O(|P| |T|)

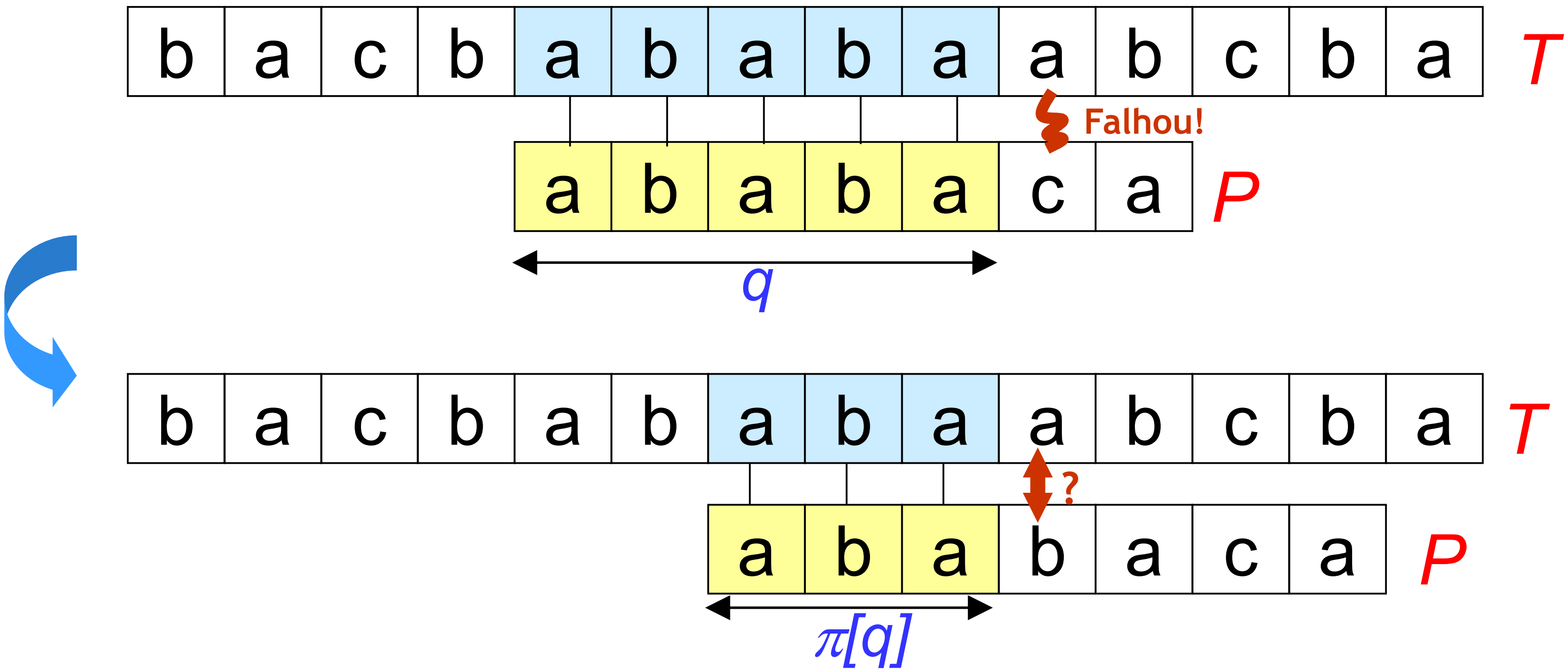
# Autómato finito correspondente ao padrão

P | a | a | b | a | a |

$\Sigma = \{a, b, c\}$



# Algoritmo de Knuth-Morris-Pratt




Desloca-se o padrão para a direita de uma forma que permite continuar a comparação na mesma posição do texto! Evita comparações inúteis!

Deslocamento é determinado por uma função $\pi[q]$ calculada numa fase de pré-processamento do padrão!

# Pré-processamento do padrão

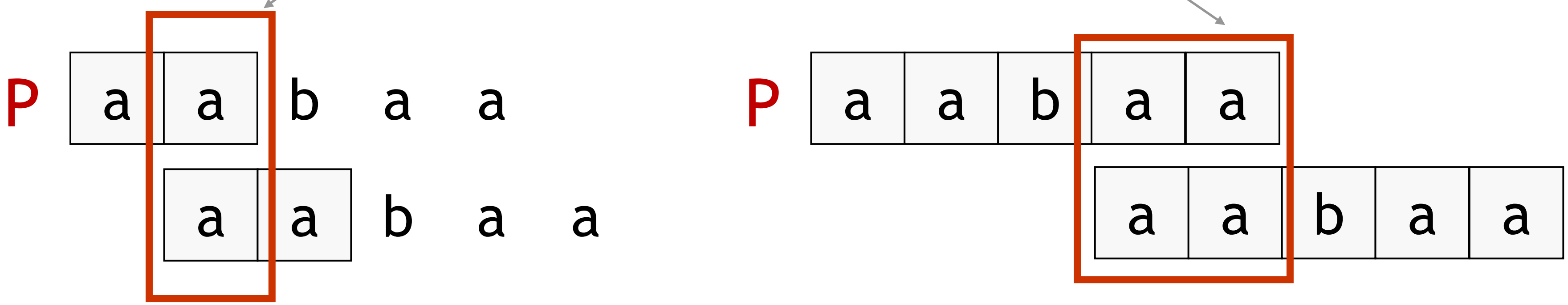
- Compara-se o padrão com deslocações do mesmo, para determinar a **função prefixo**

$$\pi[q] = \max \{k: \ 0 \leq k < q \text{ e } P[1..k] = P[(q-k+1)..q] \}$$

  - $q = 1, \ldots, |P|$
  - $P[i\ldots j]$ – substring entre índices i e j
  - Índices a começar em 1
  - $\pi[q]$ é o comprimento do maior prefixo de P que é um sufixo próprio do prefixo de P de comprimento q

# Pré-processamento do padrão

| q | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| P[q] | a | a | b | a | a |
| $\pi$[q] | 0 | 1 | 0 | 1 | 2 |



# Pseudo-código

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                                      ▷ Number of characters matched.
for i ← 1 to n                             ▷ Scan the text from left to right.
     do while q > 0 and P[q + 1] ≠ T[i]
            do q ← π[q]                    ▷ Next character does not match.
        if P[q + 1] = T[i]
          then q ← q + 1                   ▷ Next character matches.
        if q = m                           ▷ Is all of P matched?
          then print "Pattern occurs with shift" i − m
               q ← π[q]                    ▷ Look for the next match.
```

# Pseudo-código

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
    do while k > 0 and P[k + 1] ≠ P[q]
           do k ← π[k]
       if P[k + 1] = P[q]
          then k ← k + 1
       π[q] ← k
return π
```

# * Eficiência do algoritmo Knuth-Morris-Pratt

- KMP-MATCHER (sem incluir COMPUTE-PREFIX-FUNCTION)
  - Eficiência depende do nº de iterações do ciclo "while" interno
  - Dado que $0 \leq \pi[q] < q$, cada vez que a instrução 7 é executada, o valor de $q$ é decrementado de pelo menos $1$, sem nunca chegar a ser negativo
  - Dado que o valor de $q$ começa em 0 e só é incrementado no máximo $n$ vezes (+1 de cada vez, na linha 9), o nº máximo de vezes que pode ser decrementado (nas linhas 7 e 12) é também $n$

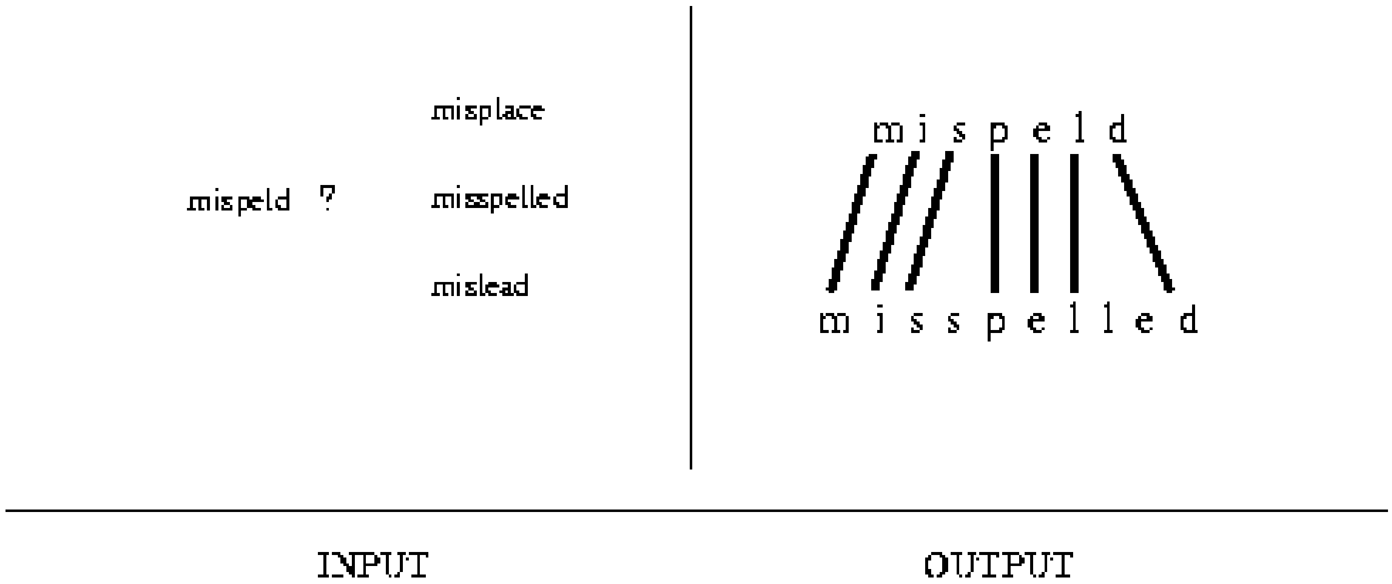
  $\Rightarrow$ Nº máximo de iterações do ciclo "while" interno (no conjunto de todas as iterações do ciclo "for" externo) é $n$

  $\Rightarrow$ Tempo de execução da rotina é $O(n)$, i.e., $O(|T|)$
- COMPUTE-PREFIX-FUNCTION
  - Seguindo o mesmo raciocínio, tempo de execução é $O(m)$, i.e., $O(|P|)$
- Total: O(n+m),isto é, **O(|T| + |P|)**

# Pesquisa aproximada

# (*approximate string matching*)

# Problema




**Input description:** A text string $T$ and a pattern string $P$. An edit cost bound $k$.

**Problem description:** Can we transform $T$ to $P$ using at most $k$ insertions, deletions, and substitutions?

(Ou: qual é o grau de semelhança entre $P$ e $T$?)

# Distância de edição entre duas strings

- A *distância de edição* entre *P* (*pattern string*) e *T* (*text string*) é o menor número de alterações necessárias para transformar *T* em *P*, em que as alterações podem ser:
  - *s*ubstituir um carácter por outro
  - inserir um carácter
  - eliminar um carácter

**P: abcdefghijkl**

**T: bcdeffghixkl**

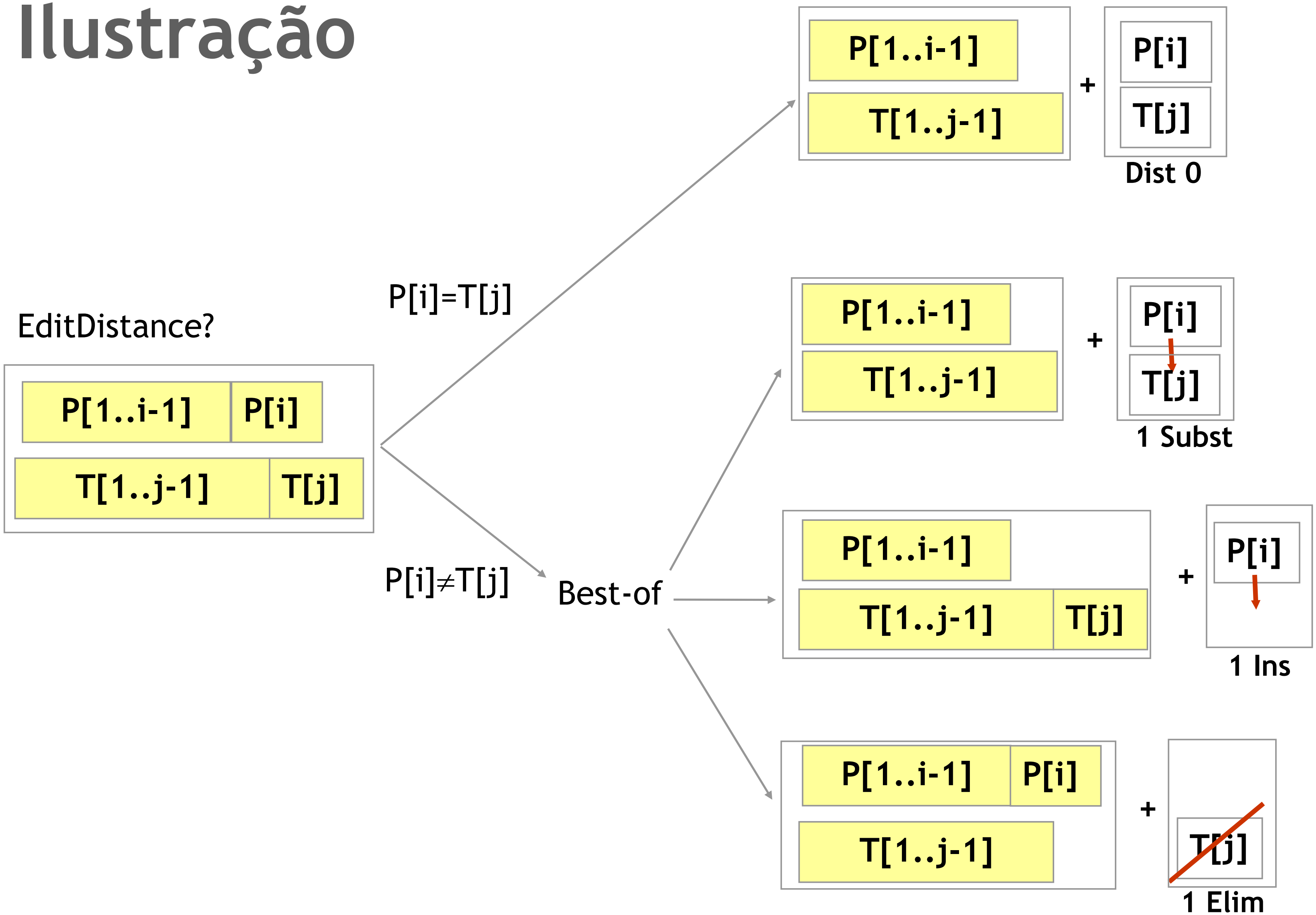
*inserção* *eliminação* *substituição*

**EditDistance(P,T)=3**

# Formulação recursiva

- $D[i,j] = EditDistance(P[1..i], T[1..j]),\ 0 \le i \le |P|,\ 0 \le j \le |T|$
- Condições fronteira:
  - $D[0, j] = j$, $D[i,0] = i$ (porquê?)
- Caso recursivo ($i>0$ e $j>0$):
  - Se $P[i]=T[j]$, então $D[i, j] = D[i-1, j-1]$
  - Senão, escolhe-se a operação de edição que sai mais barata; isto é, $D[i,j]$ é o mínimo de:
    - $1 + D[i-1, j-1]$ (substituição de $T[j]$ por $P[i]$)
    - $1 + D[i-1, j]$ (inserção de $P[i]$ a seguir a $T[j]$)
    - $1 + D[i, j-1]$ (eliminação de $T[j]$)

# Ilustração



EditDistance?
P[1..i-1]
P[i]
T[1..j-1]
T[j]
P[i]=T[j]
P[i]≠T[j]
Best-of
+
Dist 0
1 Subst
1 Ins
1 Elim

# Matriz de programação dinâmica

T

D[i,j]

P

| | | b | c | d | e | f | f | g | h | i | x | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

+1 +1 +1

# Pseudo-código

Tempo e espaço: O(|P|.|T|)

```
EditDistance(P,T) {
  // inicialização
  for i = 0 to |P| do D[i,0] = i
  for j = 0 to |T| do D[0,j] = j

  // recorrência
  for i = 1 to |P| do
    for j = 1 to |T| do
       if P[i] == T[j] then D[i,j] = D[i-1,j-1]
       else D[i,j] = 1 + min(D[i-1,j-1],
                             D[i-1,j],
                             D[i,j-1])
  // finalização
  return D[|P|, |T|]
}
```

# Optimização de espaço

Espaço: O(|T|)

```
EditDistance(P,T) {
  // inicialização
  for j = 0 to |T| do D[j] = j  // D[0,j]
  // recorrência
  for i = 1 to |P| do
    old = D[0] // guarda D[i-1,0]
    D[0] = i   // inicializa D[i, 0]
    for j = 1 to |T| do
       if P[i] == T[j] then new = old
       else new = 1 + min(old,
                          D[j],
                          D[j-1])
       old = D[j]
       D[j] = new
  // finalização
  return D[|T|]
}
```

*Tem D[i-1,j-1]*

*Ainda tem valor anterior D[i-1,j]*

*Já tem valor da iteração corrente, i.e., D[i, j-1]*

# Outros problemas

- Sub-sequência comum mais comprida (*longest common subsequence*)
  - Formada por caracteres não necessariamente consecutivos
  - ABD ? ABCDEF (delete)
- Substring comum mais comprida (*longest common substring*)
  - Formada por caracteres consecutivos
  - ABAB (BAB) BABA (BA) ABBA -> {AB, BA} (tamanho 2)
- Compressão de texto com códigos de Huffman
- Criptografia

# Exercício

- Considere as palavras "**alice**" e "**paris**":

  - Construa a matriz de programação dinâmica para este problema e indique a distância de edição entre as duas palavras;

  - Suponha agora que não é possível substituir caracteres. Assim, a distância de edição deve ser calculada apenas com <u>inserções</u> e <u>remoções</u>. Qual a distância de edição das palavras "**alice**" e "**paris**", neste caso?

# Referências e mais informação

- "Introduction to Algorithms", Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
  - Fonte consultada para o "matching" exato
- "The Algorithm Design Manual", Steven S. Skiena, Springer-Verlag, 1998
  - Fonte consultada para o "matching" aproximado
  - Discute como se usa o cálculo da distância de edição para encontrar num texto *T* a substring que faz o melhor "match" com um padrão de pesquisa *P*