Introducción a la Computación (Matemáticos)

String Matching

Temario:

- String Matching
  - Algoritmo Naïve
  - Algoritmo de Knuth-Morris-Pratt (KMP)
  - Algoritmo de Boyer-Moore

Introducción String Matching

- ¿Qué es *string matching*?
  - Encontrar todas las ocurrencias de un patrón en un texto dado (o cuerpo de texto).
- Aplicaciones
  - En editores/procesadores de palabras/browsers.
  - Chequeo de nombre de usuario y passwod.
  - Detección de Virus.
  - Análisis de Header (encabezados) en mensajes.
  - Análisis de secuencias de ADN.
  - Búsqueda en páginas webs.

Descripción del problema

- El *texto* está en un arreglo T [0..n-1] de longitud n
- El *patrón* está en un arreglo P [0..m-1] de longitud m <<<< n
- Elementos de T y P son caracteres de un *alfabeto finito* Σ
  - Ej., Σ = {0,1} o Σ = {a, b, …, z}
- Usualmente T y P son llamados *strings* de caracteres

Descripción del problema (cont.)

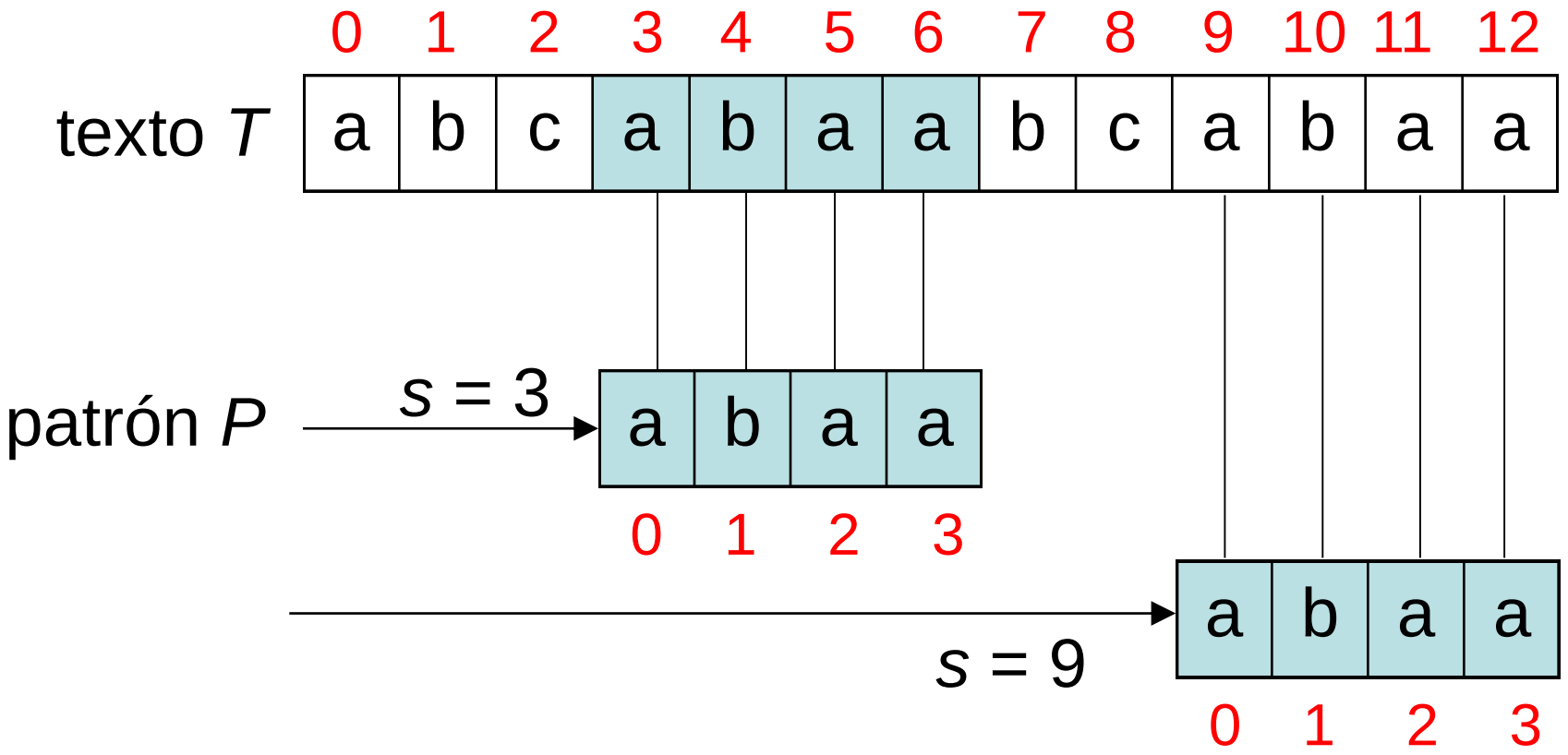
- Decimos que el patrón *P* ocurre con *shift s* en el texto *T* si:

  a) $0 \leq s \leq n-m$ y

  b) $T[(s)..(s+m)] = P[0..m-1]$

- Si *P* ocurre con shift *s* en *T*, entonces *s* es un *shift válido*, caso contrario *s* es un *shift inválido*
- Problema de String-matching : encontrar todos los shifts válidos para dados *T* y *P.*

Ejemplo:



$s = 3$ *y* $s = 9$ son shifts válidos.
(n=13, m=4 y $0 \leq s \leq n\text{-}m$ holds)

Especificación

string-matcher: $T \in Char[\,] \times P \in Char[\,] \rightarrow \mathbb{Z}[\,]$

Pre.: $\{T_0 = T \wedge P_0 = P \wedge |T_0| \geq |P_0|\}$

Pos.:$\{ \forall i \in \mathbb{Z}(0 \leq i < |RV| \; (\exists s \in \mathbb{Z}(i \leq s < |T_0| - |P_0| \wedge (RV[i] = s \leftrightarrow$
$\forall j \in \mathbb{Z}(0 \leq j < |P_0| \rightarrow T_0[k + j] = P_0[j]))) \wedge$
$\forall j \in \mathbb{Z}(i < j < |RV|) \rightarrow RV[i] \neq RV[j])\}$

Terminología

- *La concatenación* de 2 strings *x* e *y* es *xy*
  - Ejm., *x*=“sri”, *y*=“lanka” $\Rightarrow$ *xy* = “srilanka”
- Un string *w* es un *prefijo* de un string *x*, si *x=wy* para algún string *y*
  - Ejm., “srilan” es un prefijo de “srilanka”
- un string *w* es un *sufijo* de un string *x*, si *x=yw* para algún string *y*
  - Ejm., “anka” es un sufijo de “srilanka”

…. un algoritmo O(mn)

Una de las más obvias maneras de resolver el problema es comparar el primer elemento del patrón con el primer elemento del texto, si coinciden entonces comparar el segundo elemento del patrón con el segundo del texto y así seguir hasta que todo elemento del patrón es explorado. Si en alguno de los chequeos es encontrado una discordancia volver a comenzar el chequeo del primer caracter del patrón con una posición más a la derecha que en la pasada anterior y repetir todo el proceso anterior.

Algoritmo Naïve de String-Matching

Input: strings T [0..n-*1*] y P[0..m-*1*]

Result: Todos los shifts validos existentes

NAÏVE-STRING-MATCHER (T, P)

```
    n = len(T)
    m = len(P)
    for s in range(0, n-m):
      si P[0:m-1] == T [s:s+m]
            print "patron ocurre con shift" s
```

¿Como trabaja el enfoque Naïve?

Ilustremos como trabaja el algoritmo Naïve con un ejemplo.

Texto T

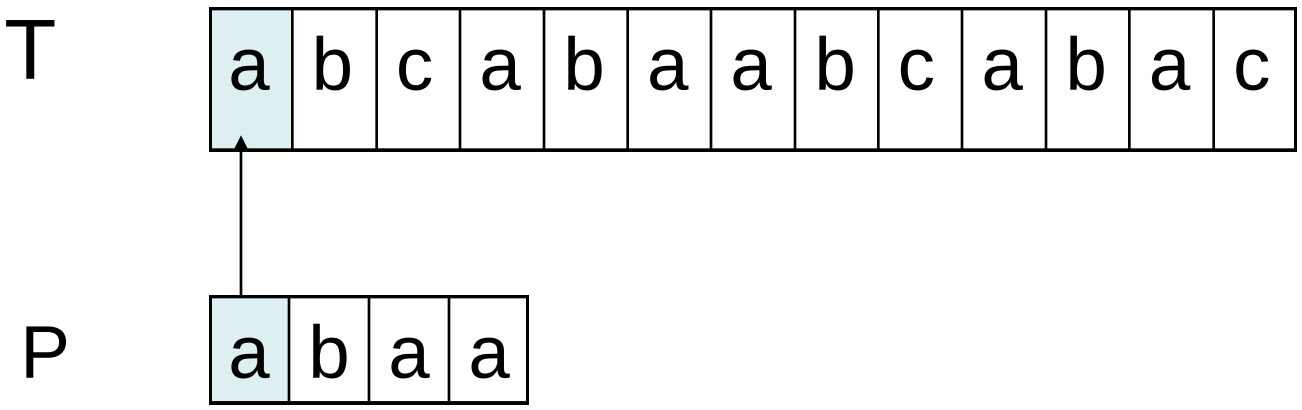
a	b	c	a	b	a	a	b	c	a	b	a	c

Patrón P

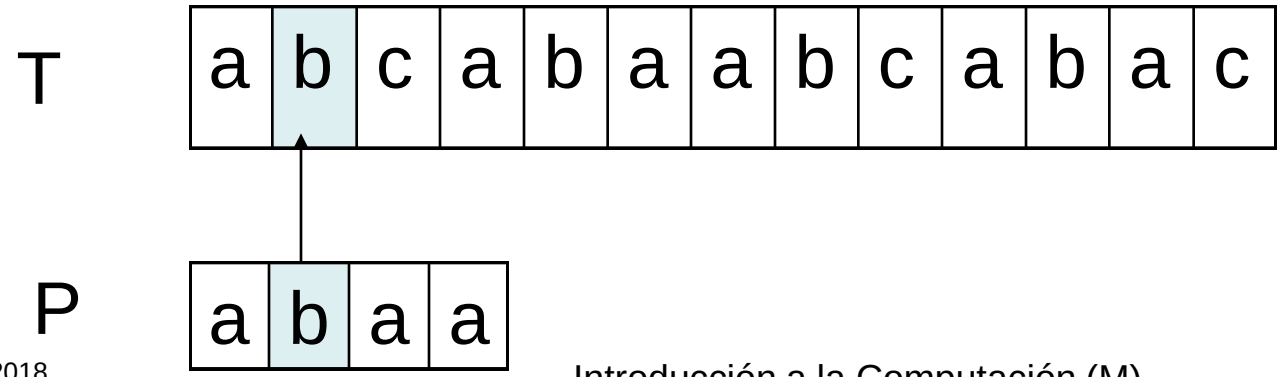
a	b	a	a

Ejemplo

1er. paso: comparar P[0] con T[0]

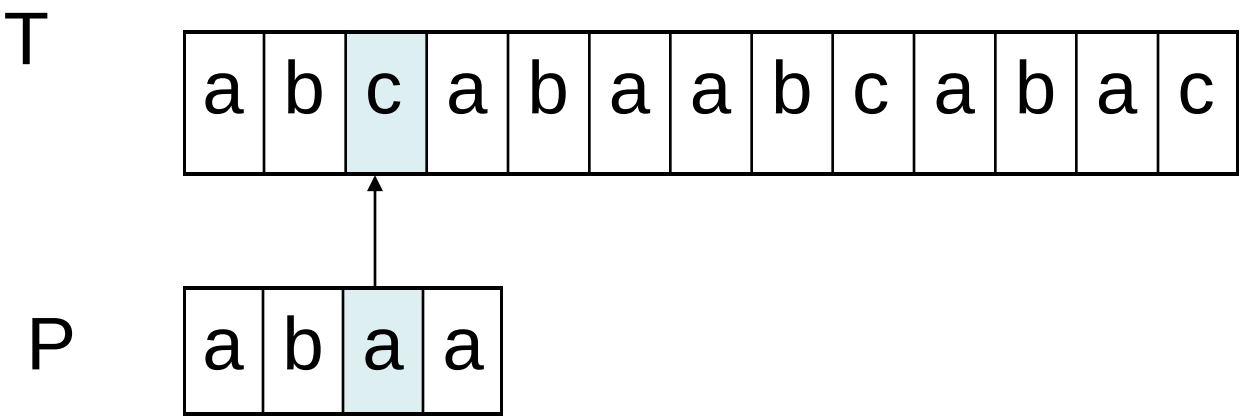


2do. paso: comparar P[1] con T[1]



Ejemplo (cont.)

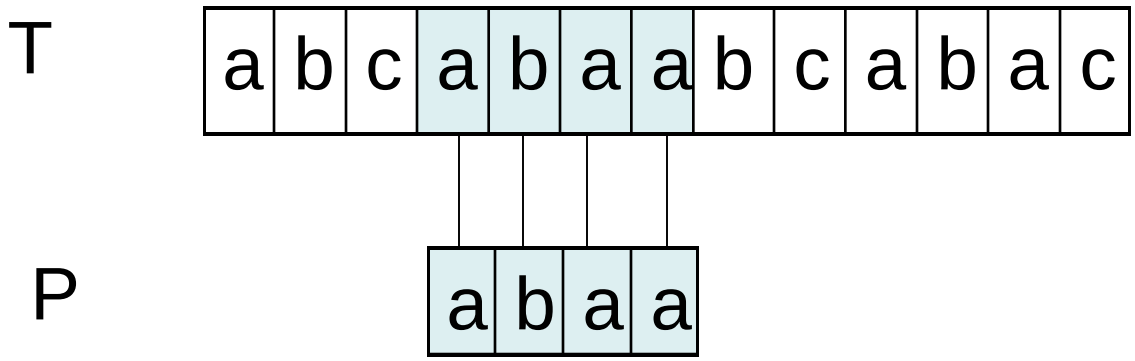
3er. paso: comparar P[2] con T[2]



un desacuerdo es encontrado aquí.

Puesto que una diferencia es encontrada, corremos ‘P’ una posición a derecha y repetimos los pasos 1 a 3. Si en alguna posición se detecta una diferencia volvemos a correr ‘P’ una posición a derecha y repetimos el procedimiento anterior.

Ejemplo (cont.)

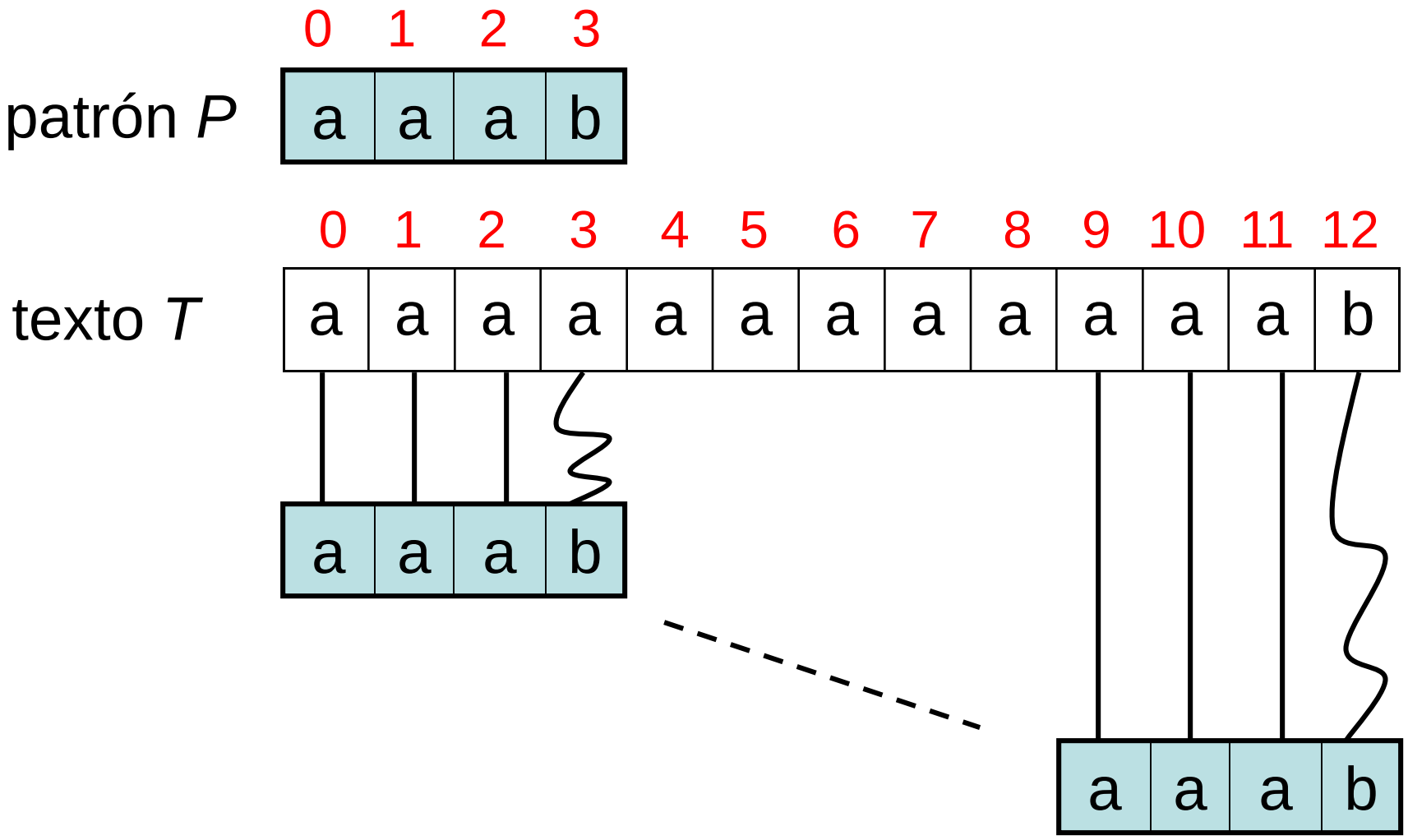


Finalmente, una correspondencia es encontrada después de desplazar tres posiciones el chequeo del patrón ‘P’ hacia la derecha. Puesto que buscamos todos los shifts válidos volvemos a empezar el procedimiento comparando P[0] y T[3].

Deficiencias

El algoritmo Naïve es ciertamente ineficiente. La razón de tal ineficiencia es debido a que los elementos del texto ‘T’ que han sido comparados en una pasada fallida son vueltos a comparar en las siguientes pasadas. Por ejemplo: cuando una discordancia es detectada la primera vez entre P[2] y T[2], ‘P’ se mueve sólo una posición a la derecha y el procedimiento de chequeo vuelve a comenzar desde allí. Sin embargo puesto que ya se sabe que T[1]=‘b’ no debería ser necesario repetir la comparación que fallará.

Análisis del peor caso:



Análisis de peor caso: (cont.)

- Hay *m* comparaciones por cada testeo en el peor caso.
- Hay *n*-*m*+1 testeos.
- Así, la complejidad temporal en el peor caso será O((*n*-*m*+1) *m*)
  - En el ejemplo anterior tendremos (13-4+1) * 4 comparaciones en total
- El método Naïve es ineficiente porque la información obtenida durante un chequeo no es utilizada para los siguientes.

Algoritmos más eficientes

- Requerirán algo de preprocesamiento, el cual redundará en un mejor comportamiento asintótico, determinístico o probabilístico.

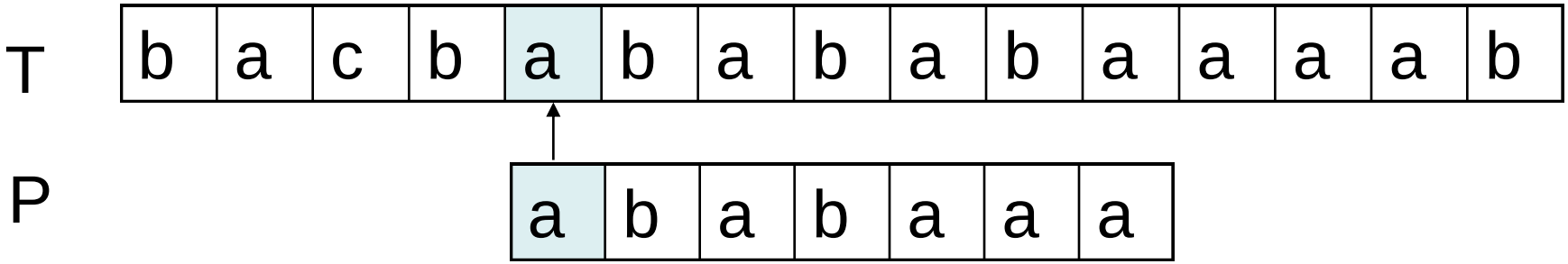
- El tiempo de preprocesamiento es usualmente dedicado a construir una función que permita una sustancial reducción en el numero de “shifts” testeados.

El Algoritmo de Knuth-Morris-Pratt

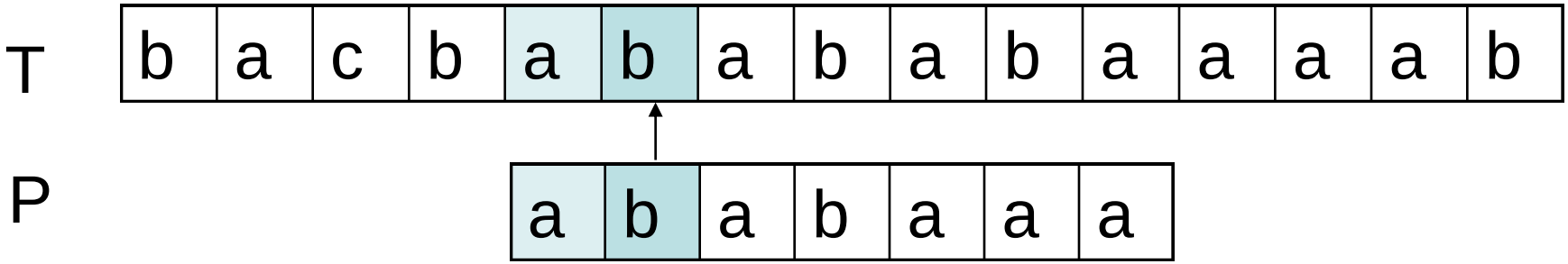
Knuth, Morris y Pratt propusieron un algoritmo de tiempo lineal para el problema de string matching.

El O(n) es obtenido evitando comparaciones de caracteres del patrón ‘P’ con elementos de ‘T’ que ya han sido analizados en comparaciones anteriores con algún otro elemento del patrón, i.e., backtracking sobre el texto ‘T’ nunca ocurre.

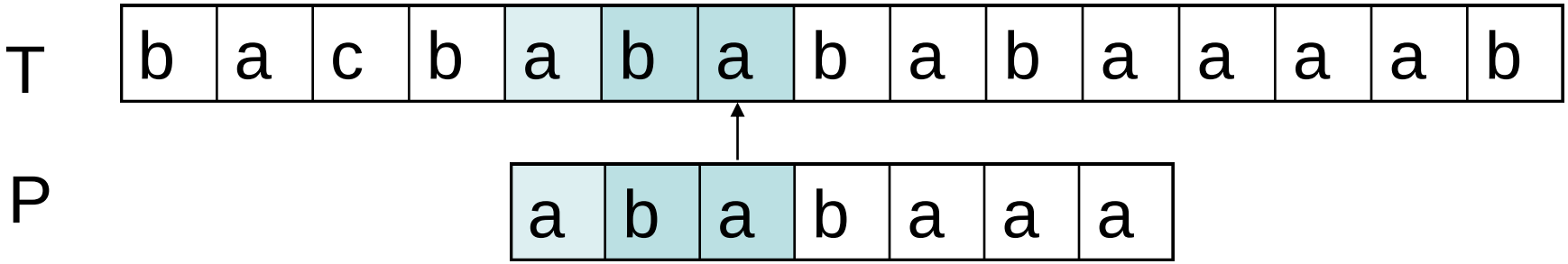
Fijemos ideas:



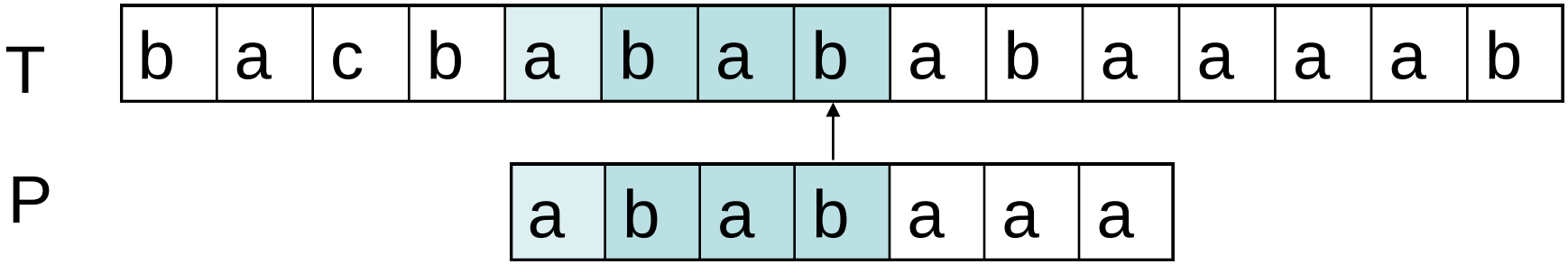
Fijemos ideas:



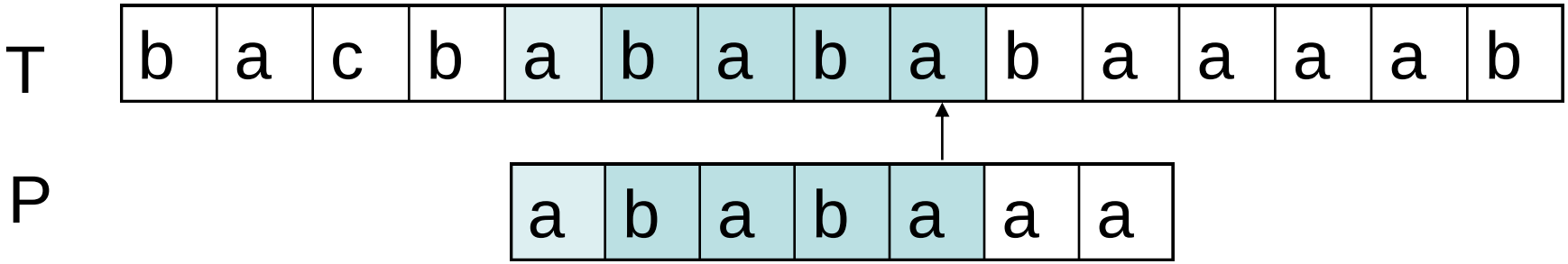
Fijemos ideas:



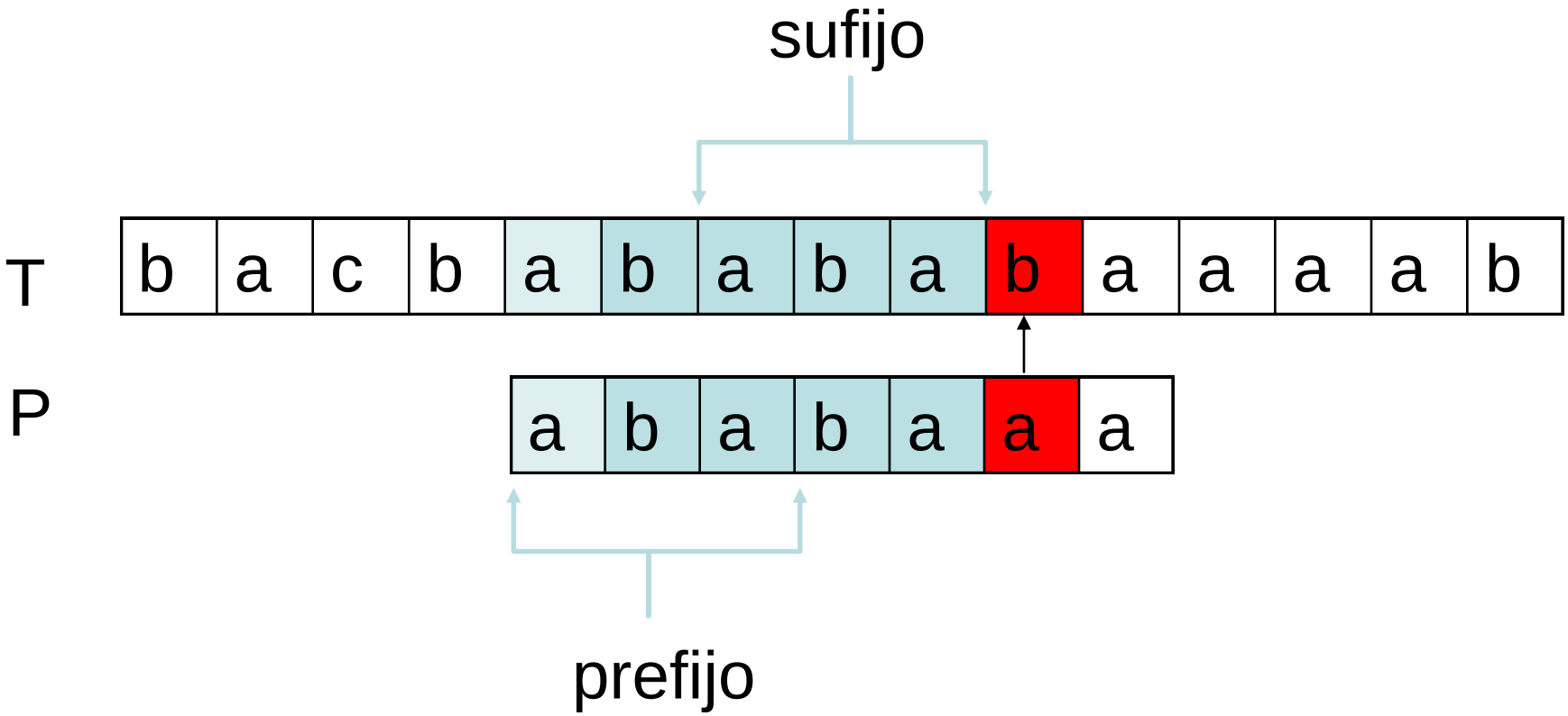
Fijemos ideas:



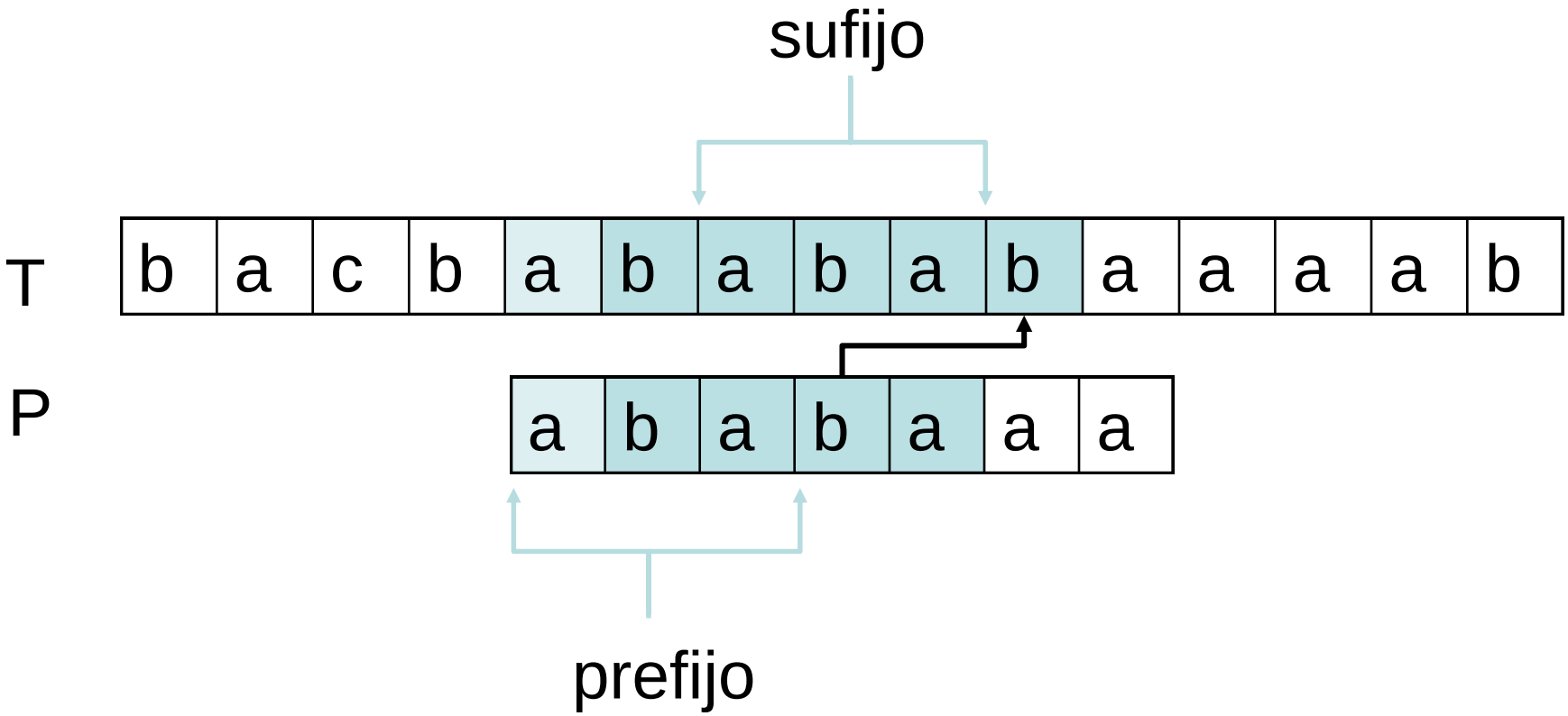
Fijemos ideas:



Fijemos ideas:



Fijemos ideas:



Componentes del algoritmo KMP

- Una función prefijo, Π

Esta función Π determina para cada prefijo del patrón la máxima cantidad de caracteres de su sufijo que hacen match contra el mismo patrón. Esa información será utilizada para evitar desplazamientos innecesarios del patrón ‘P’. En otras palabras, evitaremos backtracking sobre el texto ‘T’.

- El chequeador KMP

con el texto ‘T’y el patrón ‘P’ como inputs y la y función prefijo ‘Π’, encuentra las ocurrencias de ‘P’ en ‘T’ y retorna los valores de shifts de ‘P’ a partir de los cuales el patrón es encontrado.

Ejemplo:

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

La función prefijo, Π

```
def Compute-Prefix-Function (P):
  m = len(P)
  Π = [0]                    #Caso base.
   for q in range(1, m):
       j= Π [-1]             #Toma el anteúltimo valor.
       while j > 0 and P[q] != P[j]:
                      j = Π[j]
       if P[q] == P[j]:
                      j += 1
       Π.append = j
   return Π
```

Ejemplo: P =

a	b	a	b	a	a	a

m = len(p) = 7
Inicialmente Π[0] = 0
k = 0

Paso 1: q = 1, k=0
Π[1] = 0

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0					

Paso 2: q = 3, k = 0,
Π[3] = 1

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1				

Paso 3: q = 4, k = 1
Π[4] = 2

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1	2			

Paso 4: q = 5, k =2

Π[5] = 3

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1	2	3		

Paso 5: q = 6, k = 3

Π[6] = 1

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1	2	3	1	

Paso 6: q = 7, k = 1

Π[7] = 1

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1	2	3	1	1

Después de iterar 6 veces, el resultado completo es: →

q	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1	2	3	1	1

El chequeador KMP

Dados el patrón ‘P’, el texto ‘T’ y la función de prefijo ‘Π’ como entrada este procedimiento encuentra toda ocurrencia de ‘P’ en ‘T’.

```
def KMP-Matcher(T,P):
    n = len(T)
    m = lengthP]
    Π = Compute-Prefix-Function(P)
    q = 0                                   #posición del caracter a chequear
    for i in range(1, n-1):                 #analiza ‘T’ de izq. a derecha
        while  q > 0 and P[q+1] != T[i]:
                q = Π[q]                    #posición que se recupera
        if P[q+1] = T[i]:
                q = q + 1                   #próxima posición a chequear
        if q = m :                          #chequeamos todo el patrón?
                print “patrón ocurre con shift” i – m
```

Ejemplo: Dados el Texto ‘T’ y patrón ‘P’ anteriores

T

b	a	c	b	a	b	a	b	a	b	a	a	a	c	a

P

a	b	a	b	a	a	a

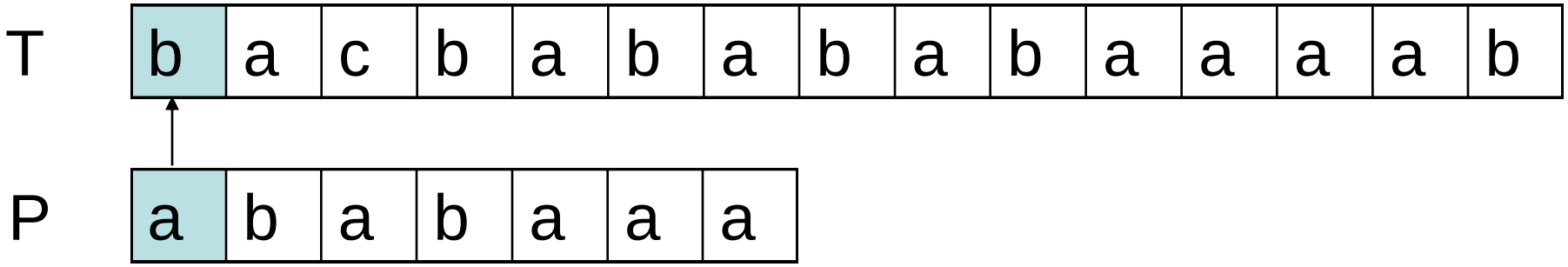
Ejecutemos el algoritmo KMP para determinar si ‘P’ ocurre en ‘T’.

La función prefijo, Π será la calculada anteriormente:

i	0	1	2	3	4	5	6
P	a	b	a	b	a	a	a
Π	0	0	1	2	3	1	1

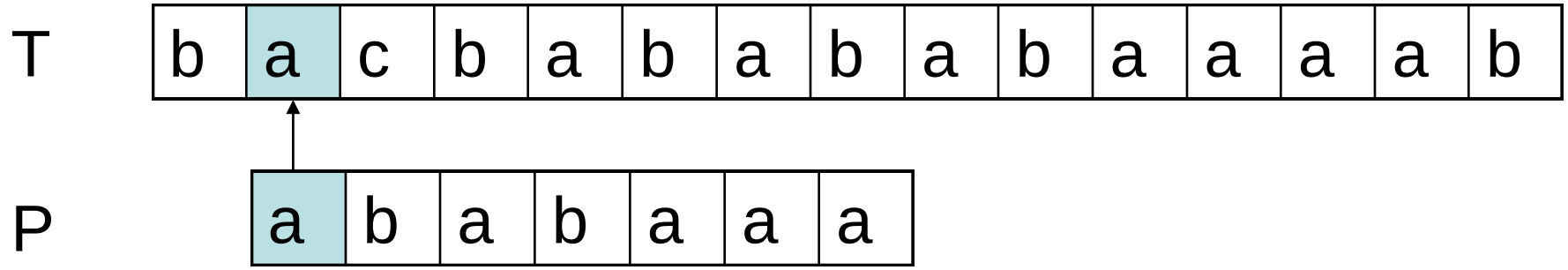
Inicialmente: n = dimensión de T = 15; m = dimensión de P = 7

Paso 1: i = 0, q = 0 comparamos P[0] con T[0]



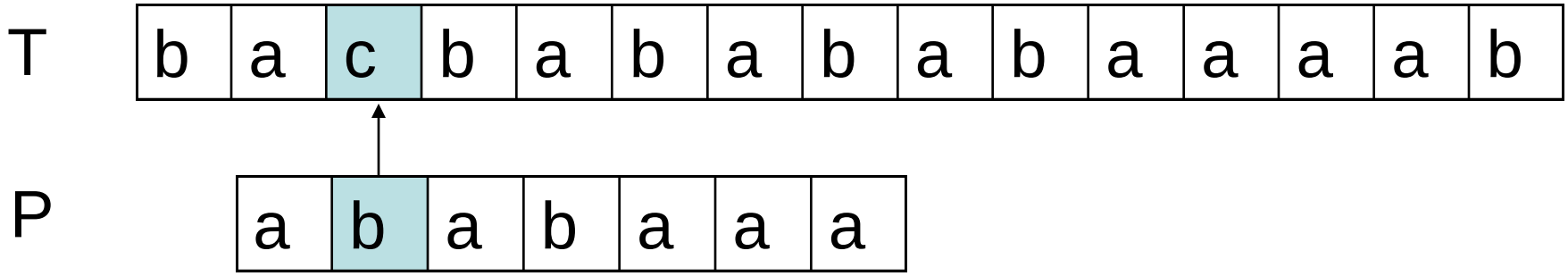
P[0] no coincide con T[0]. ‘P’ debe ser desplazado una posición a la derecha

Paso 2: 1= 2, q = 0 comparamos P[0] con T[1]



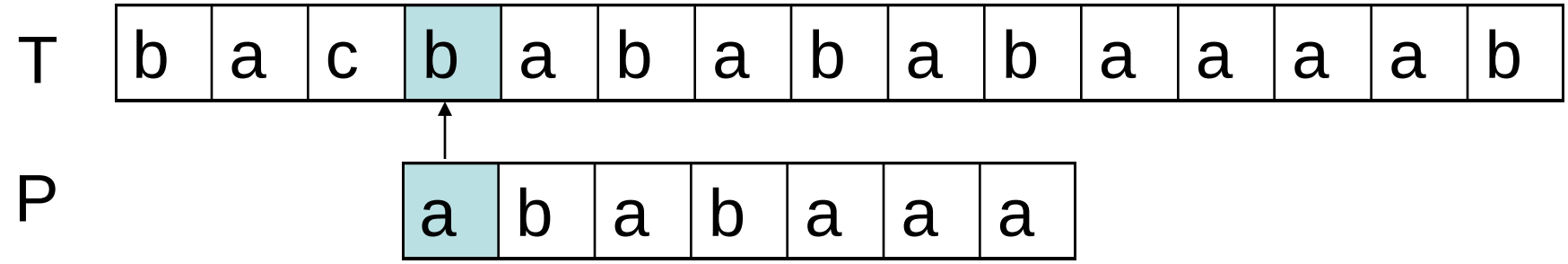
P[0] coincide con T[1]. Por lo tanto P no es desplazado.

Paso 3: i = 2, q = 1. Comparamos P[1] con T[2]. P[1] no coincide con T[2].

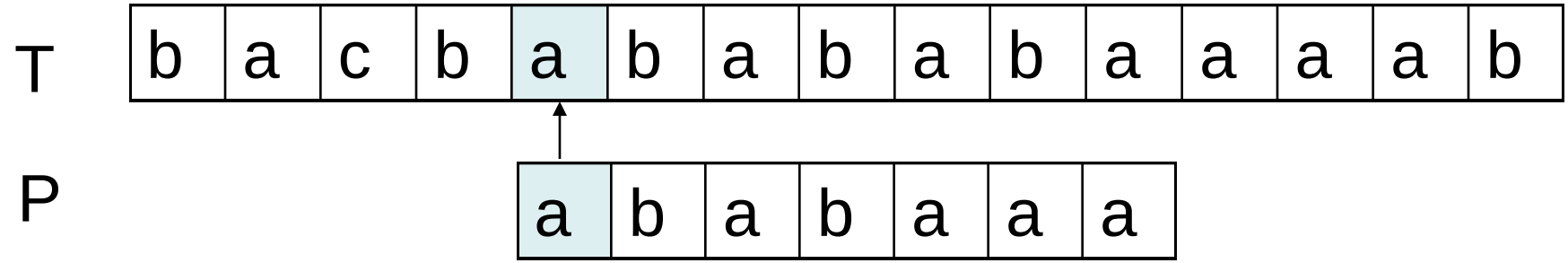


Backtracking sobre P, comparamos P[0] con T[2]

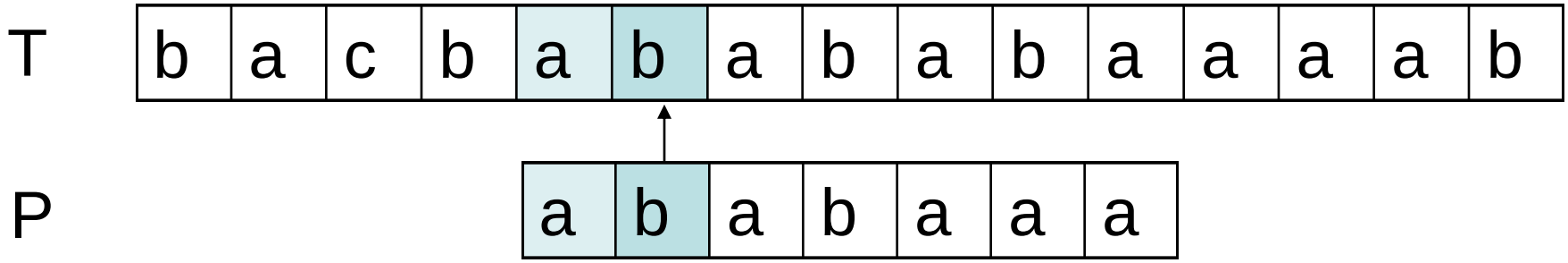
Paso 4: i = 3, q = 0. Comparamos P[0] con T[3]. P[0] no coincide con T[3].



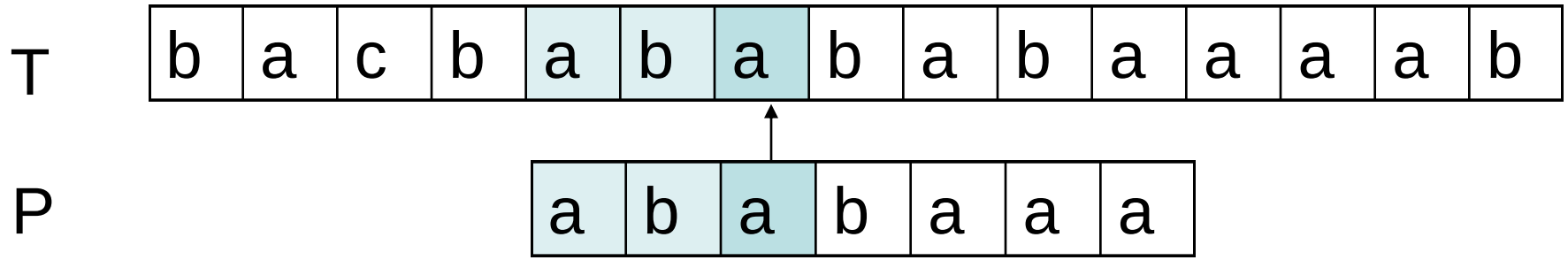
Paso 5: i = 4, q = 0. Comparamos P[0] con T[4]. P[0] coincide con T[4].



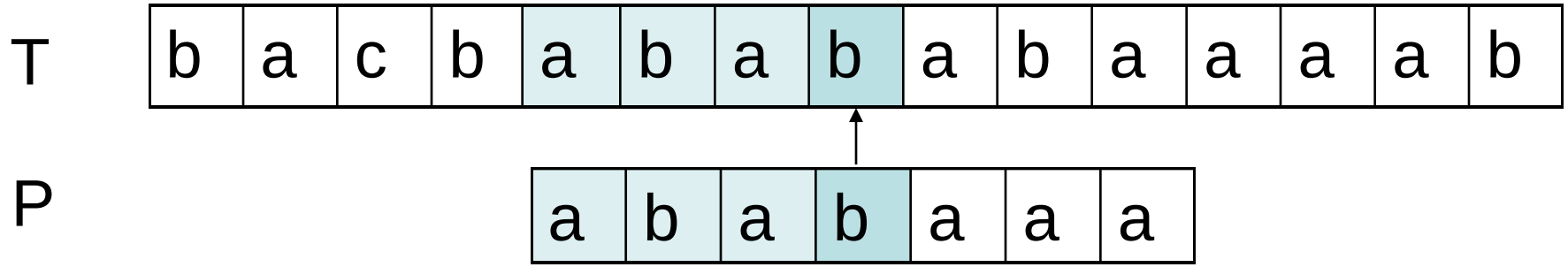
Paso 6: i = 5, q = 1. Comparamos P[1] con T[5]. P[1] coincide con T[5].



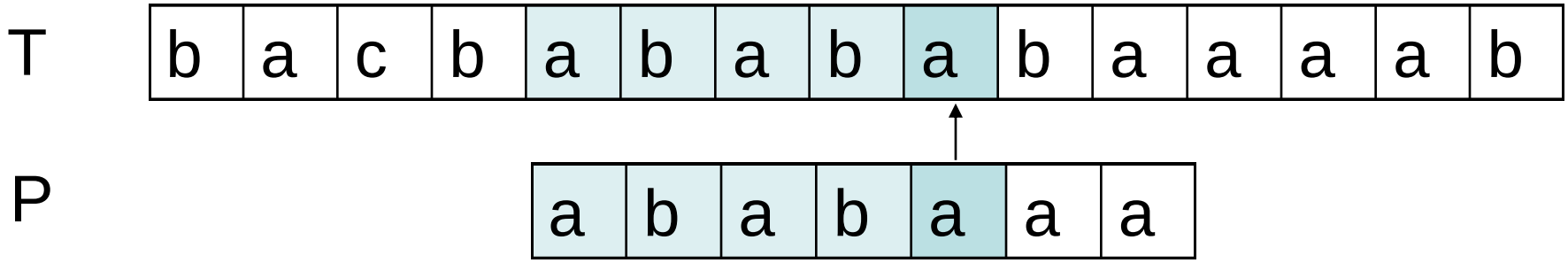
Paso 7: i = 6, q = 2. Comparamos P[2] con T[6]. P[2] coincide con T[6].



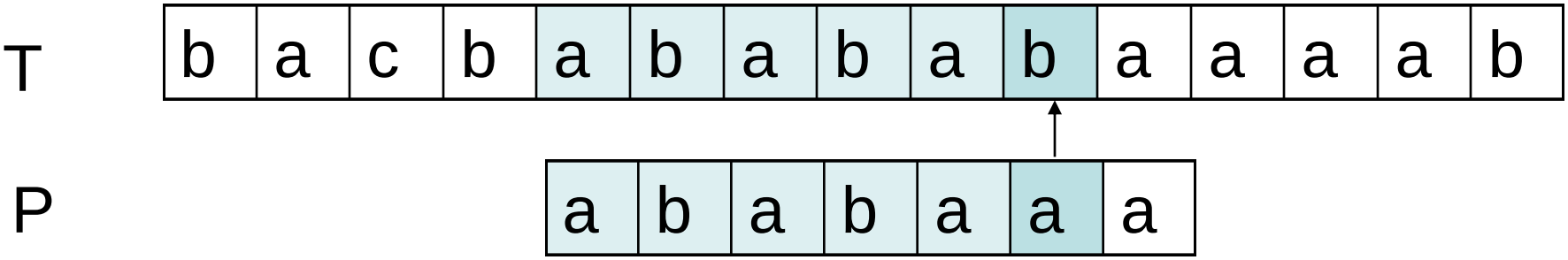
Paso 8: i = 7, q = 3. Comparamos P[3] con T[7]. P[3] coincide con T[7].



Paso 9: i = 8, q = 4. Comparamos P[4] con T[8]. P[4] coincide con T[8].

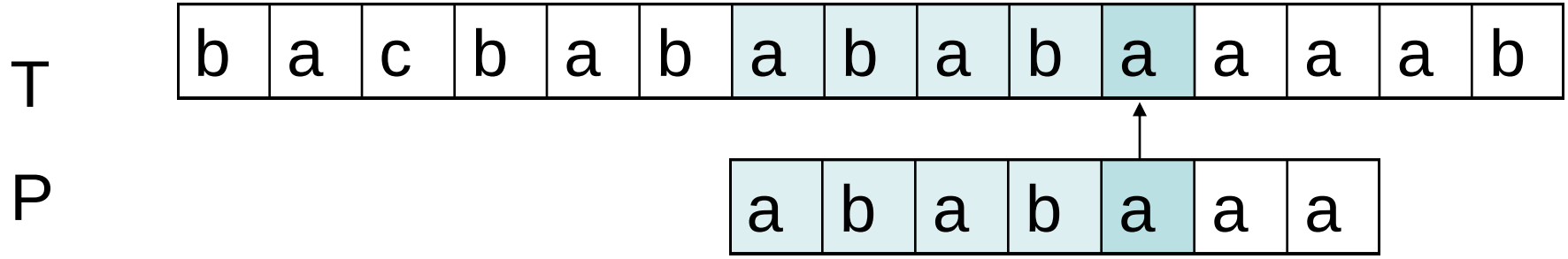


Paso 10: i = 9, q = 5. Comparamos P[5] con T[9]. P[5] no coincide con T[9].

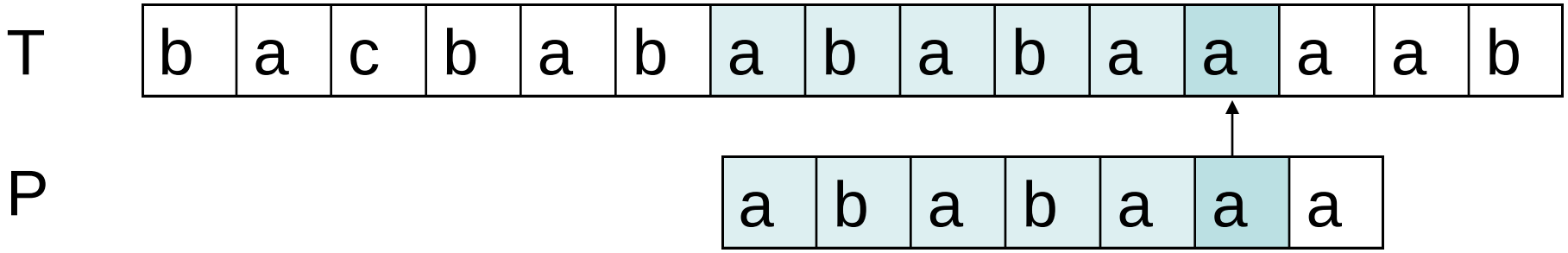


Backtracking sobre P, comparando P[3] con T[9] porque q = Π[5] = 3

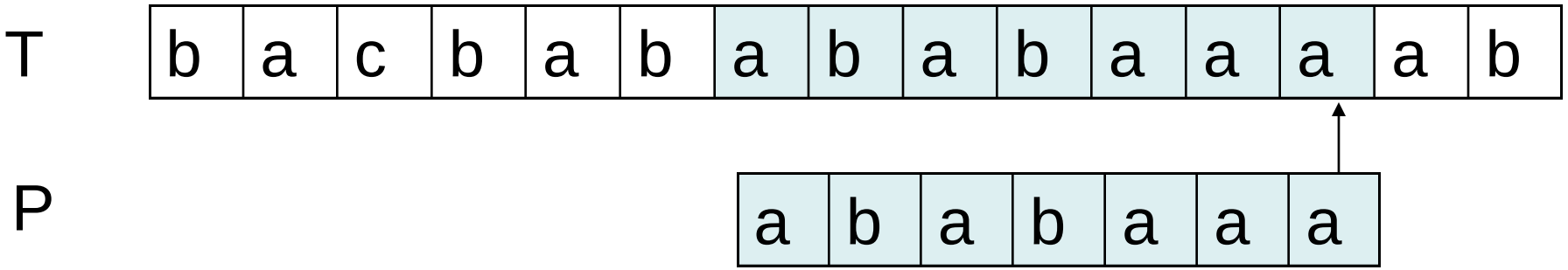
Paso 11: i = 10, q = 4. Comparamos P[4] con T[10]. P[4] no coincide con T[10].



Paso 12: i = 11, q = 5. Comparamos P[5] con T[11]. P[5] coincide con T[11].



Paso 13: i = 12, q = 6. Comparamos P[6] con T[12]. P[6] coincide con T[12].



Así el patrón ‘P’ ha sido encontrado en el texto ‘T’. El número total de desplazamientos han sido : i – m = 13 – 7 = 6 shifts.

El algoritmo de Boyer-Moore

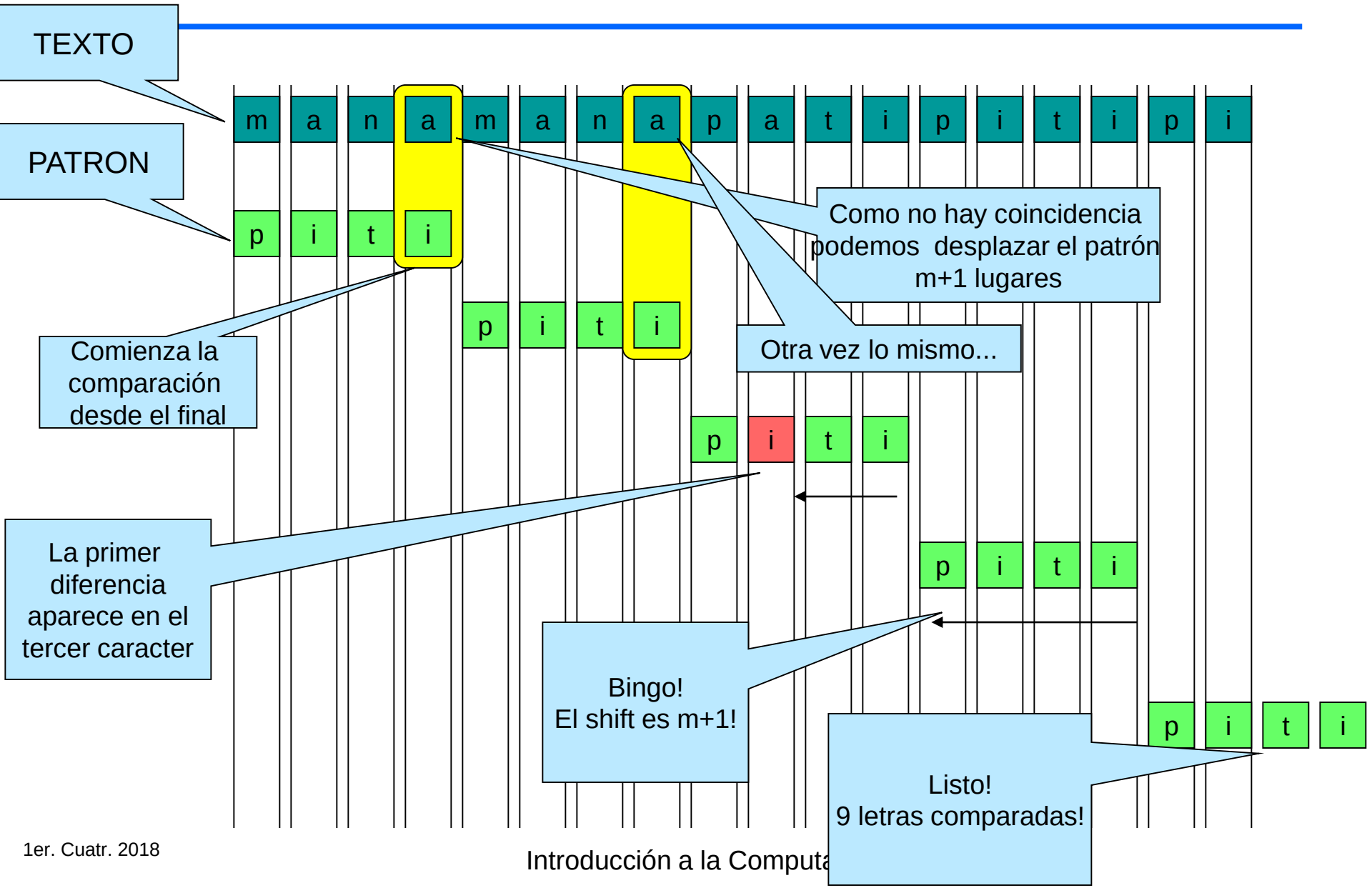
Este algoritmo es más rápido que KMP (el cual no es realmente muy rápido, aún comparado con el método de fuerza bruta) y chequea $n/m + k \bullet m$ caracteres durante el escaneo del texto, donde k es el número de ocurrencias de el patrón en el texto, siempre y cuando, asumamos que el patrón es pequeño y que el alfabeto es grande en relación a la dimensión del patrón.

El algoritmo de Boyer-Moore (cont.)

El “truco” es comparar el patrón con el texto **no** de izquierda a derecha sino de derecha a izquierda. La motivación está fundada en la idea que si el último caracter no coincide entonces podríamos dar un “salto” tan largo como el patrón mismo. Para lograr un tiempo lineal - i.e. $O(|P| + |T|)$ – construimos dos tablas:

1) Una función sufijo análoga a la prefijo de KMP,
2) Otra que por cada caracter del alfabeto me indique su última aparición en el patrón.

Boyer-Moore: la idea



Bibliografía

- “Introduction to Algorithms”. T. Cormen, Ch. Leiserson, R. Rivest y C. Stein. The Mit Press. 2001.
- “Fast pattern matching in strings”, D.E. Knuth, J.H. Morris y V.R. Pratt. *SIAM Journal on Computing*, 6(2), pp. 323-350, 1977.
- “A fast string searching algorithm”, R.S. Boyer y J.S. Moore. *Communications of the ACM*, 20(10), pp. 762-772, 1977.
- Precondicionamiento y Reconocimiento de Patrones. J.Campos.http://webdiis.unizar.es/asignaturas/EDA/ea/slides/ 7-Precondicionamiento.pdf