Búsqueda de Patrones

Prof. Cristian Cappo
abril/2018

Contenido

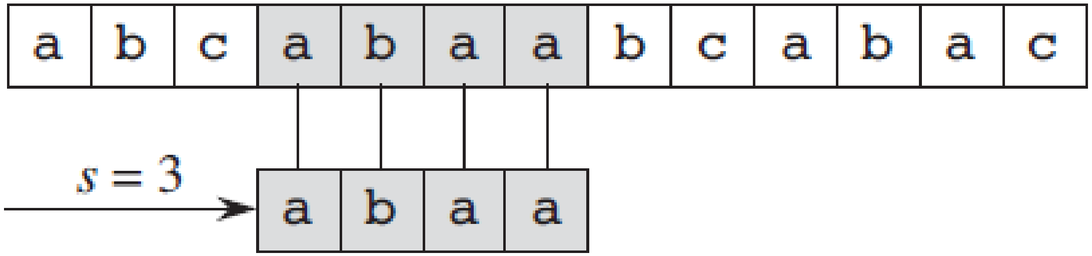
- Revisión de Conceptos
- Introducción
- Breve reseña histórica
- Algoritmos de Búsqueda de Patrones
  - Fuerza Bruta
  - K.M.P.
  - Boyer-Moore
  - Rabin-Karp

Revisión de Conceptos

- Una **cadena o *string*** es una secuencia de letras, números y caracteres especiales.
- Un tipo específico de cadena es la ***cadena binaria*** que está compuesta por secuencias de **0's y 1's**.
  - Existen aplicaciones que almacenan su información como cadenas binarias.
  - Esta distinción es necesaria, para seleccionar un algoritmo apropiado.
- El **tamaño del alfabeto** de las cadenas suele ser un **factor importante** en el **diseño de algoritmos de procesamiento de textos**. *¿Qué es **alfabeto**?*

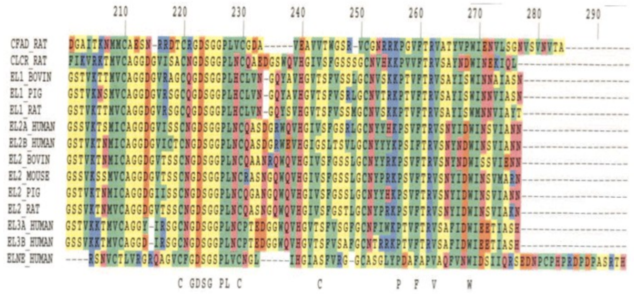
Introducción

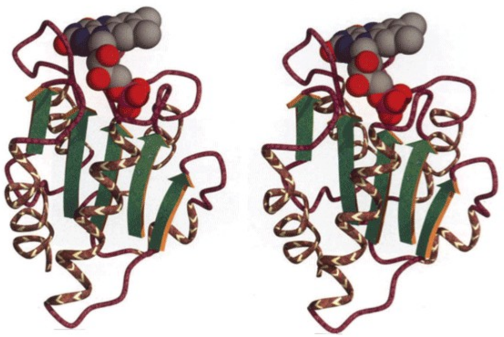
- **<u>Búsqueda de patrones (definición del problema)</u>**:
  - Asumimos que un texto es un arreglo de ***T[1..n]*** de longitud ***n*** y que el patrón es un arreglo de ***P[1..m]*** de longitud ***m*** <= n. También asumimos que los elementos de T y P son caracteres tomados de un alfabeto finito Σ , por ejemplo $\Sigma = \{0,1\}$ o $\Sigma =$ $\{$ a, b, .., z$\}$.
  - Dado un *texto* T y un *patrón* P, es encontrar una ocurrencia de dicho *patrón* dentro del *texto*.

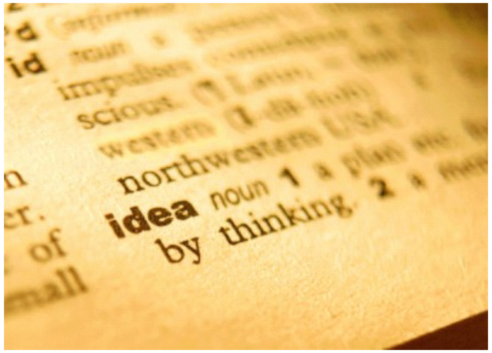


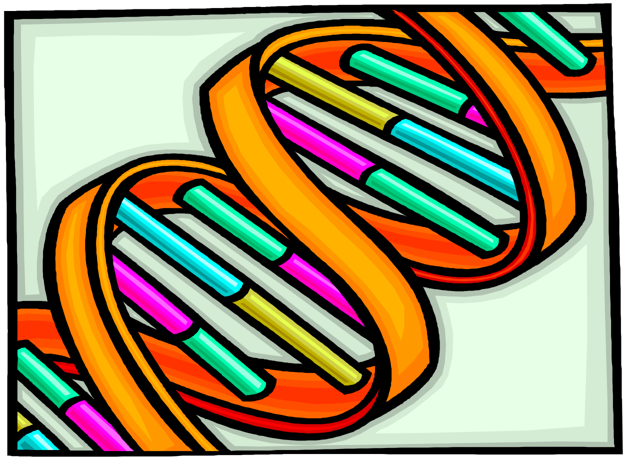
Algunas aplicaciones de la búsqueda de patrones

- Analizadores sintácticos
- Filtros de SPAM
- Librerías digitales
- Procesadores de texto
- Buscadores web
- Procesamiento de lenguaje natural
- Biología molecular computacional
- Seguridad computacional (antivirus, detección de intrusión por patrones, análisis forense, etc)
- Extracción de datos de páginas web





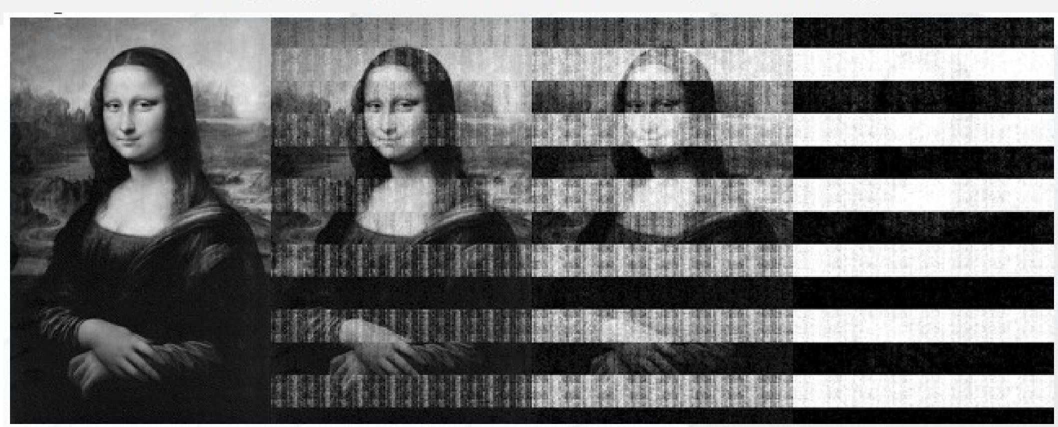




Aplicaciones

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

http://citp.princeton.edu/memory



5s 30s 60s 5m

Introducción

- Se puede ver a la **búsqueda de patrones** como una **búsqueda teniendo como clave al patrón.**
- ¿Se podrían aplicar directamente los algoritmos estudiados hasta el momento? *¿porqué*?
  - El patrón puede ser **largo**.
  - El patrón se "**alinea**" o ubica en el texto **de forma desconocida**. Es decir no hay un orden.

Terminología

- Σ^* es el conjunto de todas las cadenas formadas usando los caracteres de Σ.
- La cadena vacía es denotada por $\boldsymbol{\varepsilon}$
- La longitud de una cadena $\boldsymbol{x}$ es $|\boldsymbol{x}|$
- La cadena w es un prefijo de $\boldsymbol{x}$, simbolizado por $\boldsymbol{w} \subset \boldsymbol{x}$, si $\boldsymbol{x} = \boldsymbol{wy}$ para alguna cadena $\boldsymbol{y} \in \Sigma^*$
- La cadena $\boldsymbol{w}$ es un sufijo de $\boldsymbol{x}$, simbolizado por $\boldsymbol{w} \supset \boldsymbol{x}$, si $\boldsymbol{x} = \boldsymbol{yw}$ para alguna cadena $\boldsymbol{y} \in \Sigma^*$

Un poco de Historia

- Lo primero que se utiliza es la **fuerza bruta**, ya que es **simple y estándar.**
- Luego, en **1970**, **S. A. Cook** probó **teóricamente** la existencia de un algoritmo con tiempo $m+n$ en el peor caso.
- **D. E. Knuth** y **V. R. Pratt** siguieron laboriosamente el trabajo de Cook y obtuvieron un algoritmo que pudieron refinar en uno más simple.
- **J. H. Morris** descubrió virtualmente el mismo algoritmo de Knuth y Pratt al resolver un problema que tenía en la implementación de un editor de textos.

Un poco de Historia

- **Knuth**, **Morris** **y** **Pratt** recién en **1977** publicaron su algoritmo.
  Knuth,D.E., Morris, J. And Pratt, V. ***Fast pattern matching in strings***. SIAM J Comput. 6 (1977), 323-350.

- Para ése tiempo, **R. S. Boyer** y **J. S. Moore** descubrieron un algoritmo mucho más rápido en muchas de las aplicaciones ya que solamente **examina un fracción de los caracteres en el texto**.
  Boyer, R and Moore, S. ***A fast string searching algorithm***. *Comunications of the* ACM 20 (1977), 762-772

- El algoritmo **KMP** y el de **Boyer-Moore** necesitan un **preprocesamiento** sobre el patrón.

Un poco de historia

- En **1980**, **R. M. Karp** y **M. O. Rabin** observaron que el problema no era muy diferente de los algoritmos de búsqueda estándares y terminaron con un algoritmo que casi siempre tiene tiempo proporcional a $m+n$.

  Karp, R. and Rabin M. ***Efficient randomized pattern-matching algorithms***. Technical report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981

  Karp, R. and Rabin M. ***Efficient randomized pattern-matching algorithms***. IBM Journal of Research and Development 31 (1987), 249-260

Algoritmos resaltantes

- Fuerza Bruta
- Knuth-Morris-Pratt (KMP)
- Boyer-Moore (BM)
- Rabin-Karp
- *Y otras variantes*

Algoritmo de Fuerza Bruta

El método obvio es verificar, por cada posible posición en el texto (*t*), si el patrón (*p*) coincide o no. En este algoritmo se retorna la posición de inicio del patrón *p* en el texto *t*.

```
public static int fuerzaBruta(String p, String t) {
   int M = p.length(); N = t.length(t);
   for (int i = 0, i < N-M; i++) {
      int j
      for (j=0; j < M; j++)
        if ( t.charAt(i+j) != p.charAt(j))
           break;
      if (j == M) return i; ← retorna la pos. en t
   }
   return -1                ← retorna -1 si no encuentra
}
```

Algoritmo de Fuerza Bruta

i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B
										match		

texto = "AAAAAAAAAB"

patron = "AAAAB"

Ejemplo de peor caso

Algoritmo de Fuerza Bruta

Características

- En aplicaciones de **procesamiento de textos**, el ciclo interno raras veces es iterado, lo cual nos proporciona tiempos virtualmente proporcionales a **M+N**.
- Pero en el caso de las cadenas binarias, este algoritmo puede ser bastante lento como se vio en el ejemplo anterior (cadena de *a*'s y *b*'s).

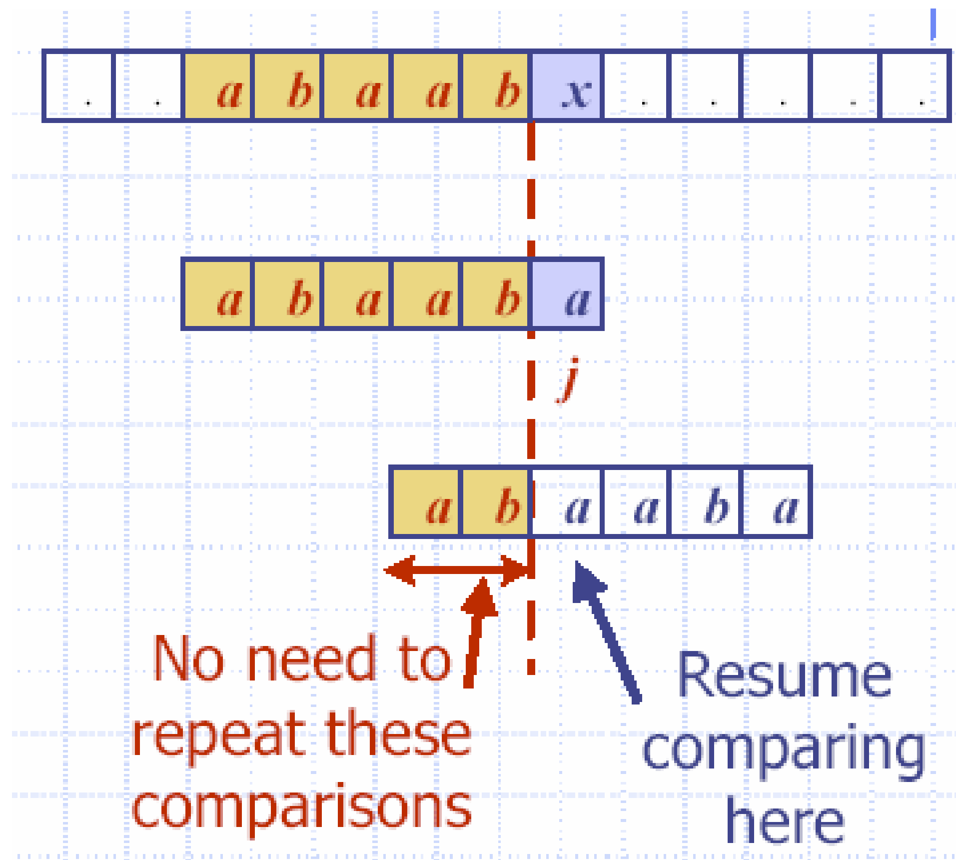
  Si el texto y el patrón está formado por 0's y terminados en 1, se da el peor caso. Se llega a un tiempo de *M(N-M+1)* y como M suele ser mucho más pequeño que N, el factor M*M es descartado quedando **M*N**.

- Problema interesante: ¿*Cómo procesar sin retroceder?*

Algoritmo de Knuth-Morris-Pratt

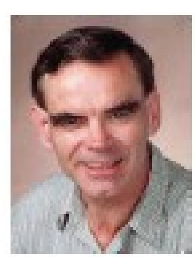
La idea principal: Si se detecta un desacierto, nuestros "inicios falsos" consisten en los caracteres que conocemos de antemano.

El **pre-procesamiento** que realiza el algoritmo genera un **arreglo** que indica **cuanto** tenemos que **retroceder en el patrón** para registrar un acierto con la posición actual en el texto.



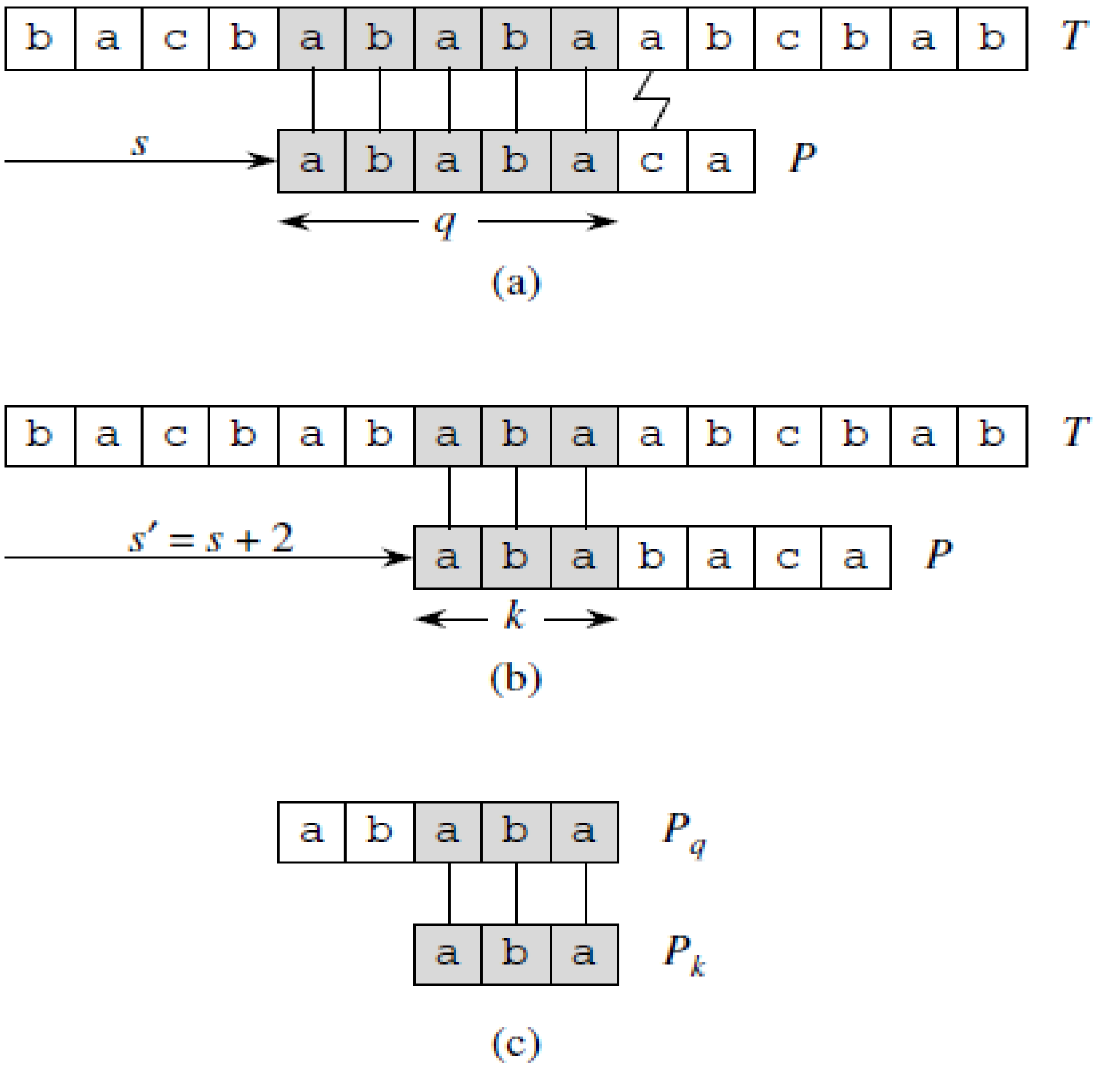
Don Knuth
1974 Turing award

Jim Morris



Vaughan Pratt

Knuth-Morris-Pratt



KMP - Preprocesamiento

- Formalización de la función de prefijo π:

$$\pi : \{1,2,..,m\} \rightarrow \{0,1,..,m-1\} \ \ tal \ \ que$$

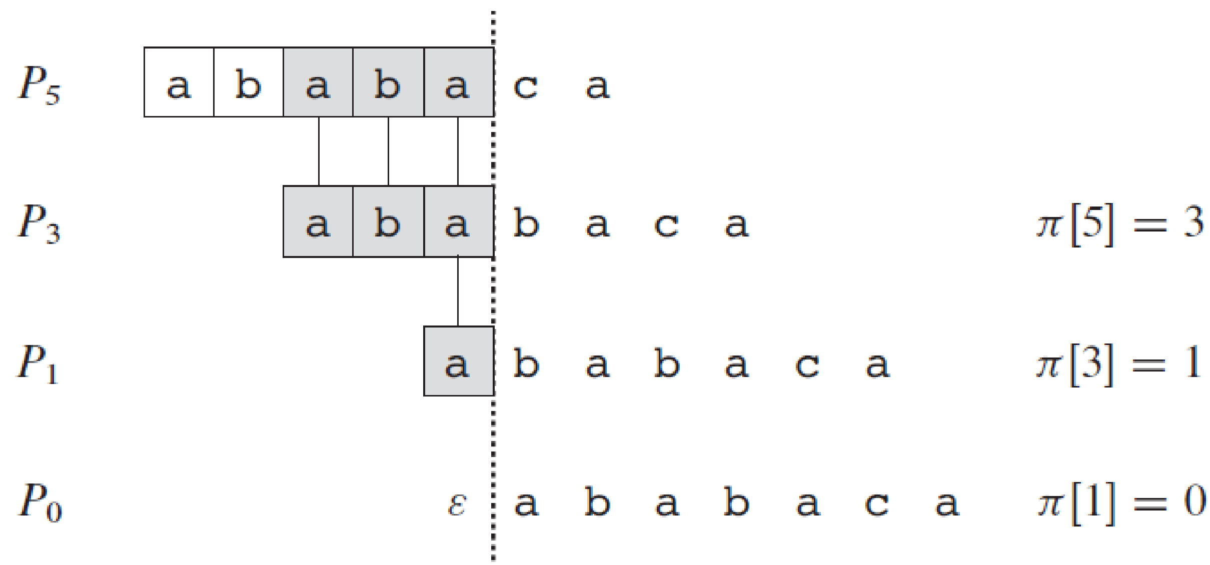
$$\pi[q] = max\{k : k < q \ \ y \ \ P_k \supset P_q\}$$

Esto es, π[q] es la longitud del prefijo más largo de *P* que es un sufijo de P_q

KMP – Preprocesamiento

- Considerando el patrón = `ababaca` y q = 5

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1



KMP – Prefix function

COMPUTE-PREFIX-FUNCTION(P)

```
m = P.length
let π[1 . . m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

KMP-CLRS

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                      // number of characters matched
for i = 1 to n                             // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                           // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                          // next character matches
    if q == m                              // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                           // look for the next match
```

Knuth-Morris-Pratt

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0
for i = 1 to n
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]
    if P[q + 1] == T[i]
        q = q + 1
    if q == m
        print "Pattern occurs with shift" i − m
        q = π[q]
```

- **Ejercicio 1:**
  - Patrón: “**1 0 1 0 0 1 1 1**”
  - Calcular el vector prefijo con el algoritmo visto anteriormente.
  - Buscar en el texto

1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1

1 0 1 0 0 1 1 1

¿Desde donde se comparará el siguiente? .. Ir completando hasta encontrar el patrón en el texto

KMP - Ejercicio

- **Ejercicio 2**
  - Aplique el algoritmo de KMP-CLRS dado el patrón y texto.

Secuencia	C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
Patrón	C	T	G	C	C	T	A	G								

- **Ejercicio 3**
  - Dado el siguiente texto: `ABAAAABAAAAAAAAA`
  - Y el patrón: `BAAAAAAAAA`

  ¿Cuántas comparaciones hizo?

Knuth-Morris-Pratt

Características

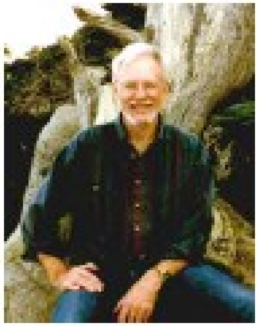
- ***KMP*** *nunca utiliza más de* ***M+N*** *comparaciones entre caracteres.*
- ***KMP*** *es más veloz cuando los desaciertos ocurren lo más tarde.*
- ***KMP*** *no trabaja eficientemente cuando el tamaño del alfabeto es grande.*

Pregunta*: ¿es necesario retroceder para continuar la búsqueda?*

Algoritmo de Boyer-Moore

- El algoritmo compara el patrón de **derecha a izquierda contra el texto dado.**
- Puede saltar **M** caracteres cuando encuentra un desacierto
- Boyer-Moore también realiza un **pre-procesamiento** generando un arreglo de saltos cuyo tamaño es proporcional al tamaño del alfabeto utilizado.
- En caso de **desacierto**, utiliza información del **carácter del texto** en el cual ocurrió el desacierto y la **posición actual** en la cual ocurrió el desacierto dentro del patrón para determinar cuántas posiciones debe saltar.

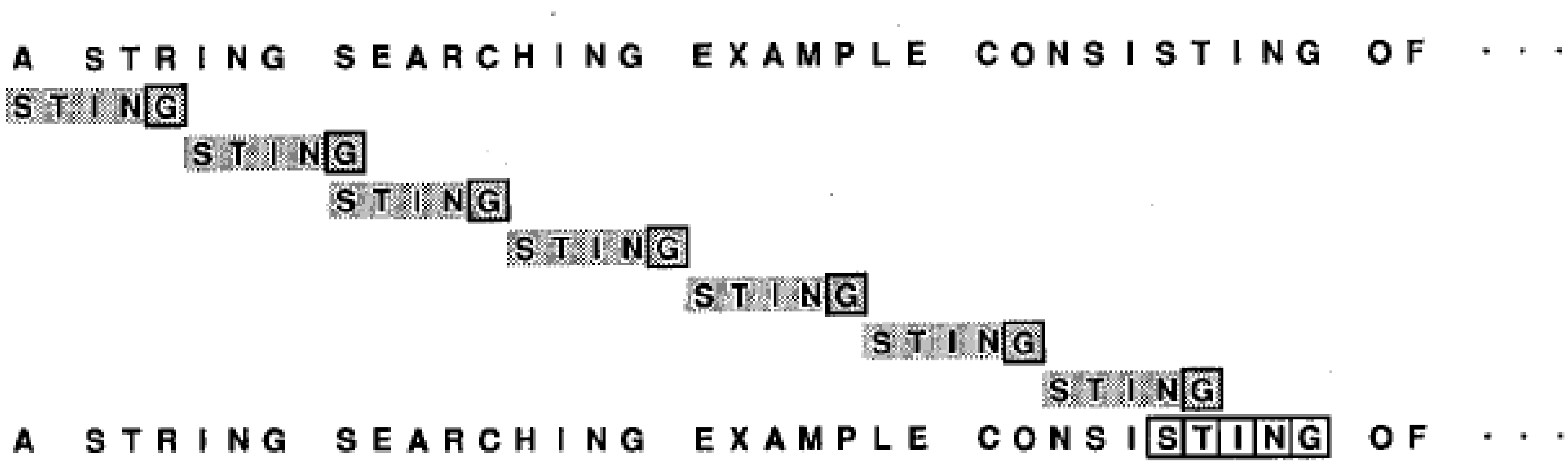
Bob Boyer



J. Strother Moore

Boyer-Moore

- Ejemplo:



Boyer-Moore

Ejemplo 2

Coincidencia

```
A   P A T T E R N   M A T C H I N G   A L G O R I T H M
R I T H M
    R I T H M
              R I T H M
                        R I T H M
                                  R I T H M
                                            R I T H M
                                              R I T H M
```

Boyer-Moore Algoritmo

```
public int search(String txt)
{
   int N = txt.length();
   int M = pat.length();
   int skip;
   for (int i = 0; i <= N-M; i += skip)
   {
      skip = 0;
      for (int j = M-1; j >= 0; j--)
      {
         if (pat.charAt(j) != txt.charAt(i+j))
         {
            skip = Math.max(1, j - right[txt.charAt(i+j)]);
            break;
         }
      }
      if (skip == 0) return i;
   }
   return N;
}
```

compute skip value

in case other term is nonpositive

match

Tabla de saltos

```
right = new int[R];
for (int c = 0; c < R; c++)
   right[c] = -1;
for (int j = 0; j < M; j++)
   right[pat.charAt(j)] = j;
```

		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

***Ejercicio:* Calcular right para** *ababaca*

Boyer-Moore

- **Ejercicio 4**

  Aplicar el algoritmo BM para la cadena

  `bacbababaabcbab`

  y el patrón

  `ababaca`

  ¿Cual fue el número de comparaciones se hicieron? Indique la tabla de saltos.

Boyer-Moore

- *El algoritmo de **Boyer-Moore** nunca utiliza más de **M+N** comparaciones entre caracteres.*
  - En el **peor caso** el algoritmo puede ser ***NM***.
  - El **caso medio** es de **N/M** y se da en modelos de cadenas aleatorias, las cuales son casi siempre no-realistas.
- El algoritmo no ayuda mucho si se lo utiliza con cadenas binarias ya que el alfabeto queda reducido a 2 opciones únicamente.

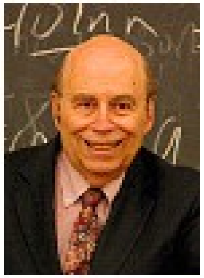
KMP vs. Boyer-Moore

- Boyer-Moore
  - Alfabetos grandes
  - Entrada aleatoria
  - Patrones largos
  - Ejemplo: sentencias en español o inglés
- KMP
  - Alfabetos pequeños
  - Datos estructurados
  - Ejemplo: genoma

Algoritmo de Rabin-Karp

- Una **aproximación a la búsqueda de fuerza bruta** sería tratar a cada posible sección de M-caracteres como una clave de una **tabla hash**.
- El método se basa en calcular la **función hash** para la posición *i* dado el valor para la posición *i-1*.
- **`h(x) = x mod q`**
  - q es un número primo grande.
  - $\mathbf{x = a[i]d^{M-1} + a[i+1]d^{M-2} + \ldots + a[i+M-1]d^{0}}$
- La siguiente posición **`i+1`** se calcula:
  - $\mathbf{x = (x - a[i]d^{M-1})d + a[i+M]}$

Dick Karp
1985 Turing award



Michael Rabin

Rabin-Karp

Ex. Hash "table" size = 97.

Pattern				
5	9	2	6	5

59265 % 97 = 95

Text																				
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	2	3	8	4	6

3	1	4	1	5					31415 % 97 = 84
	1	4	1	5	9				14159 % 97 = 94
		4	1	5	9	2			41592 % 97 = 76
			1	5	9	2	6		15926 % 97 = 18
				5	9	2	6	5	59265 % 97 = 95

Rabin-Karp

- Calcular el *hash* puede hacerse en O(m) operaciones por *hash*.
- Método más rápido:
  - Pre-calculado : 10000 % 97 = 9
  - Cálculo previo : **4**1592 % 97 = 76
  - Siguiente hash : 1592**6** % 97 = ??
- Observación *(truco matemático)*
  - 15926 % 97 $= (\mathbf{4}1592 - (\mathbf{4} * 10^4) * 10 + \mathbf{6}$
    $= (76 - (\mathbf{4} * 9)) * 10 + \mathbf{6}$
    $= 406 \% 97$
    $= 18$

$10^4 \bmod 97$

Rabin-Karp

```
public static int rksearch ( String p, String t)
  int q = 33554393   // tamaño de la tabla
  int d = 256        // radix (el numero de la base)
  int M = p.length(); int N = t.length();

  int dM = 1        // calcular d^(M-1) % q

  for (int i = 1; i < M; i++)
     dM = (d*dM) % q;

  for (i = 0; i < M; i++) {
    h1 = (h1*d+p.charAt(i)) % q;  //hash patron
    h2 = (h2*d+t.charAt(i)) % q;  //hash texto
  }
  if ( h1 == h2 ) return 0;  // encontre en el cero

  for (i = M; i < M ; i++) {
    h2 = (h2 + d*q - dM*t.charAt(i-M)) % q; //sacar 1ro
    h2 = (h2*d + t.charAt(i)) % q; //poner ultimo
    if (h1 == h2) return i-M +1 ;  //se encontró
  }
  return -1;                          //no se encontró
}
```

Rabin-Karp

- La búsqueda de patrones por el método de Rabin-Karp es extremadamente lineal.
- Pueden ocurrir coincidencias a las cuales llamaremos aciertos fallidos o espúreos que ocurre cuando 2 secciones distintas de longitud M arrojan el mismo valor de clave hash. Si colocamos esta verificación necesitamos backup (Abordaje : *Las Vegas). Sin* backup es una *versión Montecarlo* con la ventaja de no requerir backup pero no asegura la correctitud.
- Teóricamente el algoritmo aún podría tomar tiempo M*N en el peor caso (que es prácticamente imposible). Si q es un primo suficientemente grande (cerca de MN^2) entonces la probabilidad de colisión de 1/N.

Ejercicios adicionales

Ejercicio 5
Dado q=11, cuantos aciertos espurios se producen en el texto T=`3141592653589793` cuando se busca el patrón P=26.

Ejercicio 6
En la fase de precálculo de KMP, se le pasó una cadena P que produjo la siguiente tabla `{0 0 1 2 3 4 5 6 0 1}.` Invente tres posibles cadena para P.

Ejercicio 7
Cuantas comparaciones hará el algoritmo de BM para buscar cada uno de los siguientes patrones en un texto binario de 1000(mil) ceros?

`a)  00001` `b) 10000`

Referencias

- T. Cormen, C. Leiserson, R. Rivest y C. Stein. *Introduction to algorithms*. Second edition. MIT Press. 2001. Capítulo 32.

- Robert Sedgewick. *Algorithms*. Addison Wesley. 1983. *String searching, chapter 19.*

- Robert Sedgewick & Kevin Wayne. *Algorithms. 4th Edition. Addison Wesley. 2011. Chapter 5 (5.3 Substring search.)*

- Curso COS226 en Princeton (ver *substring search*) http://www.cs.princeton.edu/courses/archive/spring18/cos226/