

Algoritmo KMP

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

12-2021

Contenido

Manejo de cadenas de caracteres

- Introducción

- Función Z

 - Implementación ingenua (fuerza bruta)

 - Implementación eficiente

- Problema de la coincidencia de un patrón

 - Coincidencia de un patrón (fuerza bruta)

 - Algoritmo KMP (Knuth-Morris-Pratt)

Introducción

- ▶ El manejo de texto involucra una gran cantidad de información, por lo que es evidente la necesidad de automatizar ciertas tareas del manejo de texto. Funciones tales como búsqueda de un cadena, búsqueda y reemplazo de palabras y detección de errores ortográficos, entre otros.
- ▶ En años recientes, la biología molecular, al representar el genoma como una secuencia de caracteres llamados bases (A de adenina, C de citosina, G de guanina y T de timina), se ha enfrentado a la necesidad de realizar el tratamiento y análisis de grandes cantidad de cadenas de caracteres.

- ▶ Un cadena de caracteres (o string) es una secuencia ordenada de caracteres. El primer carácter en la cadena es el más izquierdo y ocupa la posición 1, el siguiente la posición 2 y así sucesivamente.
- ▶ La longitud de un cadena S , $|S|$, está dada por el número de caracteres que contiene.

- ▶ Una subcadena (o substring) de la cadena S es cualquier cadena de caracteres que se encuentran en S . La subcadena $S[i..j]$ es la cadena formada por los caracteres de S que están de la posición i a la j , inclusive.
- ▶ Prefijo: dada una cadena S , un prefijo $S[1..i]$ del mismo es cualquier subcadena que inicia en la posición 1 y termina en la posición i .
- ▶ Sufijo: dada una cadena S , un sufijo $S[i..n]$ del mismo es cualquier subcadena que inicia en la posición i y termina en el último elemento de S .

Función Z

Dada una cadena S de longitud n , la función Z es un arreglo de longitud n donde el i -ésimo elemento es igual al máximo número de caracteres empezando desde la posición i que coincide con los primeros caracteres de S . En otras palabras, $Z[i]$ es la longitud del prefijo común más largo entre S y el sufijo de S que empiezan en i , $S[i..n]$.

Veamos un ejemplo. Dada la cadena $S = \text{"aaabaaab"}$, para cualquier $i > 1$,

► Posición 2,

$$\begin{aligned} S &= \text{aaabaaab} \\ S[2..n] &= \text{aabaaab} \end{aligned}$$

► Posición 3,

$$\begin{aligned} S &= \text{aaabaaab} \\ S[3..n] &= \text{abaaab} \end{aligned}$$

► Posición 4,

$$\begin{aligned} S &= \text{aaabaaab} \\ S[4..n] &= \text{baaab} \end{aligned}$$

string	a	a	a	b	a	a	a	b
posición	1	2	3	4	5	6	7	8
Valores Z	0	2	1	0	4	2	1	0

Función Z

Procedure 1 Z_FUNCTION

```
Input: S : String  
        Z : Array[S.length]  
        INIT(Z, 0)  
        for i  $\leftarrow$  1 to S.length do  
            for j  $\leftarrow$  i to S.length do  
                if S[i] = S[j - i] then  
                    Z[i]  $\leftarrow$  Z[i] + 1  
                else  
                    break  
                end if  
            end for  
        end for  
        return Z
```

Este algoritmo tiene una complejidad $O(n^2)$. Para obtener un algoritmo eficiente, calcularemos los valores de $Z[i]$ de 1 a $(n - 1)$, pero al mismo tiempo, al calcular un nuevo valor, intentaremos hacer el mejor uso posible de los valores anteriormente calcular. Es decir, vamos a usar memorización. Y, bueno, recuerda que estamos buscando el máximo prefijo. ¿Qué técnica podríamos emplear para este algoritmo?

Implementación eficiente

- ▶ Llamemos “coincidencias de segmento” a las subcadenas que coinciden con el prefijo de S . Por ejemplo, el valor de $Z[i]$ es la longitud de la coincidencia de segmento que comienza en la posición i y que termina en la posición $i + Z[i] - 1$.
- ▶ Para hacer esto, mantendremos los índices $[l, r]$ del segmento coincidente más derecho. Es decir, entre todos los segmentos detectados mantendremos el que termina más a la derecha. En cierto modo, el índice r puede verse como el límite de lo que se ha registrado. Todo lo que está más allá aún no se conoce.

Entonces, si el índice actual, i , tenemos dos opciones:

- ▶ $i > r$: La posición está fuera de lo que ya hemos procesado. Entonces, no nos queda más utilizar la comparación carácter a carácter que vimos en el algoritmo ingenuo. Sin embargo, al final, si $Z[i] > 0$, tendremos que actualizar los índices del segmento más derecho, porque está garantizado que el nuevo $r = i + Z[i] - 1$ es menor que el r anterior.

- ▶ $i \leq r$: La posición está dentro del segmento de coincidencias actual $[l, r]$.
 - ▶ Entonces podemos usar los valores Z ya calculados para inicializar el valor de $Z[i]$ a algo (mejor que comenzar desde cero), tal vez incluso un número más grande.
 - ▶ Para ello, observamos que las subcadenas $S[l..r]$ y $S[0..(r-l)]$ coinciden. Eso significa que como aproximación inicial para $Z[i]$ podemos tomar el valor ya calcular del segmento correspondiente $S[0..(r-l)]$, que es $Z[i-l]$.
 - ▶ Sin embargo, el valor $Z[i-l]$ podría ser demasiado grande: cuando se aplica a la posición i podría exceder el índice r . Esto no está permitido porque no sabemos nada sobre los caracteres a la derecha de r , ya que pueden diferir de los requeridos. Veamos un ejemplo de este punto. Si $S = "aaaabaa"$. Cuando llegamos a la última posición ($i = 6$), el segmento de coincidencia actual será $[5, 6]$, para el cual el valor de la función Z es $Z[1] = 3$. Obviamente, no podemos inicializar $Z[6]$ en 3, sería completamente incorrecto. El valor máximo al que podríamos inicializarlo es 1, porque es el valor más grande que no nos lleva más allá del índice r . Por lo tanto, como una buena aproximación, podemos decir que $Z[i] = \min(r - i + 1, Z[i-l])$. Ahora solo falta calcular las posiciones que se encuentran a la derecha de r , y para ello usamos el algoritmo previo.

Procedure 2 Z_FUNCTION_DP

Input: S : String

Z : Array[S.length]

left \leftarrow 0, right \leftarrow 0, INIT(Z, 0)

for i \leftarrow 1 **to** S.length **do**

if i \leq right **then**

 Z[i] \leftarrow MIN(right - i + 1, Z[i - left])

end if

while i + Z[i] < S.length **and** S[Z[i]] == S[i + Z[i]]

do

 Z[i] \leftarrow Z[i] + 1

end while

if i + Z[i] - 1 > right **then**

 left \leftarrow i, right \leftarrow i + Z[i] - 1

end if

end for

return Z

string	a	a	a	b	a	a	a	b
posición	1	2	3	4	5	6	7	8
Valores Z	0	2	1	0	4	2	1	0

Aplicaciones¹

- ▶ Búsqueda de una subcadena (o patrón). (SPOILER: ¡Es el siguiente algoritmo!)
- ▶ Número de subcadenas distintas en una cadena.
- ▶ Comprensión de cadenas.

¹<https://bit.ly/3rDn1uZ>

Problema de la coincidencia de un patrón

El problema de la coincidencia de un patrón en un texto (Pattern Matching Problem) también conocido como búsqueda de una subcadena en un texto (Substring Search), se define de la siguiente forma:

- Dada una cadena P , llamada patrón (pattern) y una cadena más grande T llamada el texto, encontrar la primer ocurrencia del patrón P en el texto T y regresar la posición del texto T donde inicial el patrón o -1 en caso de que el patrón no existe en el texto T .

Por ejemplo, si $P = \text{"aca"}$ y $T = \text{"bacacabcaca"}$, el patrón P se encuentra en T iniciando en las posiciones 2, 4 y 9. Observa que las ocurrencias pueden estar sobre puestas tal y como sucede en las ocurrencias que inicial en 2 y 4. Para este caso, la respuesta es 2, debido a que es la posición de la primera ocurrencia de P en T .

La forma más simple de resolver este problema es por fuerza bruta, el cual consiste en ir moviendo a la derecha (shifting) una ventana que contiene el patrón, llamada ventana de deslizamiento (sliding windows), casilla por casilla sobre la cadena T y, en cuanto se encuentre la coincidencia, se regresa el número de la casilla de inicio.

Coincidencia de un patrón

Procedure 3 PATTERN_MATCHING

Input: P, T : String

```
for i ← 1 to T.length - P.length do
  if P = T[i..(i + P.length)] then
    return i
  end if
end for
return -1
```

La complejidad del algoritmo anterior es $O(nm)$. Sin embargo, existen varias formas de mejorar este algoritmo para hacerlo en tiempo lineal. Una de ellas es el uso de la función Z, calcula en tiempo lineal. Para lograrlo, basta con concatenar el patrón al inicio del texto, separados por un carácter que no esté en el texto. Normalmente se utiliza el signo de pesos (\$).

- ▶ Por ejemplo, si $P = \text{"aca"}$ y $T = \text{"bacacabcaca"}$, haremos una nueva cadena T' formado con la concatenación de P y T , separados por un \$.
- ▶ Después de este proceso, las ocurrencias estarán indicadas en el arreglo de la función Z por las posiciones que tengan un valor de Z igual a la longitud del patrón y restándole la longitud del patrón más 1 (por el \$).

string	a	c	a	\$	b	a	c	a	c	a	b	c	a	c	a
posición	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Valores Z	0	0	1	0	0	3	0	3	0	1	0	0	3	0	1

Algoritmo KMP (Knuth-Morris-Pratt)

- ▶ Una mejor forma de resolver el problema de la coincidencia de un patrón es utilizar el algoritmo KPM, propuesto por Donal Knuth, Jame Morris y Vaughan Pratt en 1977.
- ▶ Recordemos que la forma ingenua resuelve el problema utilizando una ventana deslizante que se recorre un carácter a la vez. Lo que busca el algoritmo KPM es que, cuando se tenga una NO coincidencia, se recorran tantos caracteres como sea necesario siempre que se garantice que el patrón no estará ahí.

La idea básica es:

- ▶ Se van comparando los caracteres a partir de la posición i del texto con los del patrón.
- ▶ Cuando uno coincide, en la subcadena del patrón que se tiene antes la NO coincidencia, buscamos el máximo prefijo que al mismo tiempo sea sufijo de la subcadena.
- ▶ Si no existe, nos regresamos solo a la posición inicial de patrón.
- ▶ Si existe un prefijo que es sufijo, y la longitud de este prefijo es k , continuamos a partir de la posición $i + k$ del texto y $j + k$ del patrón.
- ▶ Este proceso se hace tan hacia atrás como sea necesario, pudiendo llegar a inicial el patrón a su posición inicial.

Por ejemplo, si parte del texto es $T = \text{"abcabx..."}$, y el patrón es $P = \text{"abcaby"}$, iniciamos comparando los caracteres uno a uno, hasta que llegamos a la posición 6. En esta posición falla porque tenemos una x en el texto y una y en el patrón. En el método ingenuo tendríamos que iniciar todo desde la posición 2 del texto y la 1 del patrón. Pero, podemos observar que en la subcadena anterior a la falla, es decir, en "abcab" , existe el prefijo ab , con longitud 2, que al mismo tiempo es sufijo de la misma cadena ($abcab$), lo que nos dice qué es seguro que los 2 caracteres anteriores al fallo coinciden completamente con el inicio del patrón, por lo que las siguientes comparaciones se pueden hacer a partir de la 6 del texto y la posición 3 del patrón.

Para poder hacer todo lo anterior en tiempo lineal, es necesario hacer un pre-procesamiento del patrón. Este pre-procesamiento se hace usando un arreglo que inicial en la posición 0 (que sería la posición inicial de la cadena). Al final de este proceso, el arreglo contiene la longitud del máximo prefijo que es también sufijo en la subcadena $P[1..i]$.

Procedure 4 PREPROCESSING

Input: P : String

V : Array[P.length]

$j \leftarrow 0, i \leftarrow 0, V[0] \leftarrow 0$

while $i < P.length$ **do**

if $P[i] = P[j]$ **then**

$V[i] \leftarrow j + 1, i \leftarrow i + 1, j \leftarrow j + 1$

else

if $j = 0$ **then**

$V[i] \leftarrow 0, i \leftarrow i + 1$

else

$j \leftarrow V[j - 1]$

end if

end if

end while

return V

Índices	j	i							
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0								

(a)

Índices	j			i					
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0						

(c)

Índices		j				i			
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0	1				

(e)

Índices				j				i	
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0	1	2	3		

(g)

Índices			j						i
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0	1	2	3	1	

(i)

Índices	j		i						
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0							

(b)

Índices	j			i					
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0					

(d)

Índices			j				i		
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0	1	2			

(f)

Índices	j							i	
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0	1	2	3		

(h)

Índices			j						i
Patrón	a	b	c	x	a	b	c	a	b
Posición	0	1	2	3	4	5	6	7	8
Valores	0	0	0	0	1	2	3	1	2

(j)

- ▶ Una vez preprocesado el patrón, el algoritmo es muy simple dado que el arreglo V no dice que, una vez hallado un fallo, a qué posición se debe regresar el patrón para seguir comparando con el texto.
- ▶ La complejidad del algoritmo resultante es lineal sobre la longitud del texto, $O(n)$. Y como la complejidad del preprocesamiento es lineal sobre la longitud del patrón, $O(m)$, la complejidad final del algoritmo KMP es lineal sobre los procesos, $O(n + m)$, mucho más eficiente que el $O(nm)$ del algoritmo ingenuo.

Procedure 5 KMP

Input: P, T : String

$V \leftarrow \text{PREPROCESSING}(P)$, $i \leftarrow 0$, $j \leftarrow 0$, $\text{pos} \leftarrow 0$

while $i < T.\text{length}$ **do**

if $T[i] = P[j]$ **then**

$i \leftarrow i + 1$, $j \leftarrow j + 1$

if $j = P.\text{length}$ **then**

return pos

end if

else

if $j = 0$ **then**

$i \leftarrow i + 1$, $\text{pos} \leftarrow i$

else

$\text{pos} \leftarrow i - V[j - 1]$, $j = V[j - 1]$

end if

end if

end while

return -1