Búsqueda en texto

La búsqueda de patrones en un texto es un problema muy importante en la práctica. Sus aplicaciones en computación son variadas, como por ejemplo la búsqueda de una palabra en un archivo de texto o problemas relacionados con biología computacional, en donde se requiere buscar patrones dentro de una secuencia de ADN, la cual puede ser modelada como una secuencia de caracteres (el problema es más complejo que lo descrito, puesto que se requiere buscar patrones en donde ocurren alteraciones con cierta probabilidad, esto es, la búsqueda no es exacta).

En este capítulo se considerará el problema de buscar la ocurrencia de un patrón dentro de un texto. Se utilizarán las siguientes convenciones:

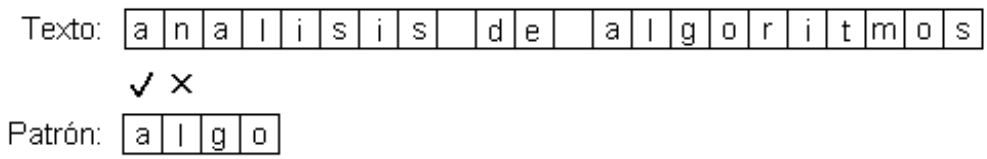
- *n* denotará el largo del texto en donde se buscará el patrón, es decir, *texto* = $a_1\, a_2 \ldots a_n$.
- *m* denotará el largo del patrón a buscar, es decir, *patrón* = $b_1\, b_2 \ldots b_m$.

Por ejemplo:

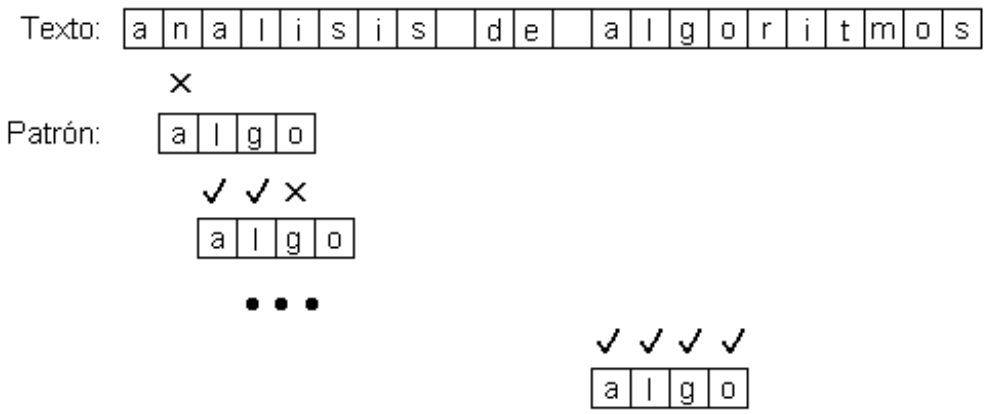
- Texto = "analisis de algoritmos".
- Patrón = "algo".

Algoritmo de fuerza bruta

Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, esto es, se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.



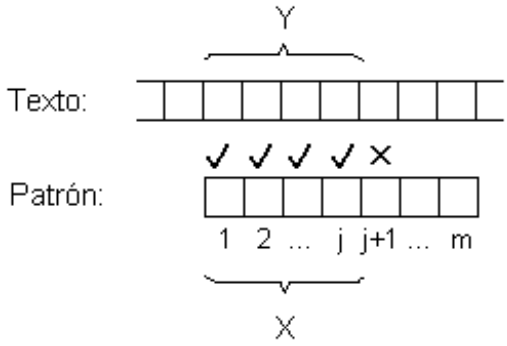
Si se detiene la búsqueda por una discrepancia, se desliza el patrón en una posición hacia la derecha y se intenta calzar el patrón nuevamente.



En el peor caso este algoritmo realiza $O(mn)$ comparaciones de caracteres.

Algoritmo Knuth-Morris-Pratt (KMP)

Suponga que se está comparando el patrón y el texto en una posición dada, cuando se encuentra una discrepancia.



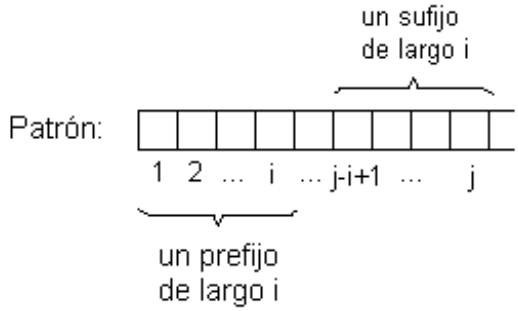
Sea *X* la parte del patrón que calza con el texto, e *Y* la correspondiente parte del texto, y suponga que el largo de *X* es *j*. El algoritmo de fuerza bruta mueve el patrón una posición hacia la derecha, sin embargo, esto puede o no puede ser lo correcto en el sentido que los primeros *j-1* caracteres de *X* pueden o no pueden calzar los últimos *j-1* caracteres de *Y*.

La observación clave que realiza el algoritmo Knuth-Morris-Pratt (en adelante KMP) es que *X es igual a Y*, por lo que la pregunta planteada en el párrafo anterior puede ser respondida mirando solamente el patrón de búsqueda, lo cual permite precalcular la respuesta y almacenarla en una tabla.

Por lo tanto, si deslizar el patrón en una posición no funciona, se puede intentar deslizarlo en 2, 3, ..., hasta *j* posiciones.

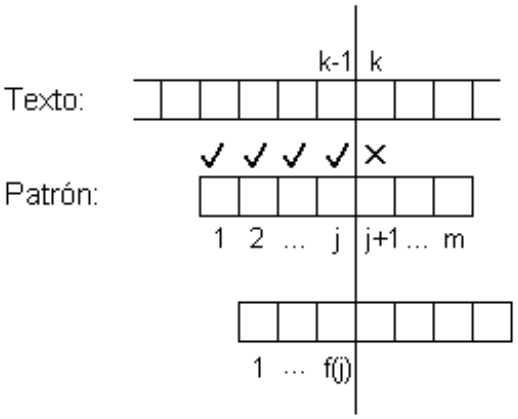
Se define la *función de fracaso* (failure function) del patrón como:

$$f(j) = max(i < j \mid b_1 .. b_i = b_{j-i+1} .. b_j)$$



Intuitivamente, *f(j)* es el largo del mayor prefijo de *X* que además es sufijo de *X*. Note que *j = 1* es un caso especial, puesto que si hay una discrepancia en b_1 el patrón se desliza en una posición.

Si se detecta una discrepancia entre el patrón y el texto cuando se trata de calzar b_{j+1}, se desliza el patrón de manera que $b_{f(j)}$ se encuentre donde b_j se encontraba, y se intenta calzar nuevamente.



Suponiendo que se tiene *f(j)* precalculado, la implementación del algoritmo KMP es la siguiente:

```
// n = largo del texto
// m = largo del patron
// Los indices comienzan desde 1

int k=0;
int j=0;
while (k<n && j<m)
{
  while (j>0 && texto[k+1]!=patron[j+1])
  {
    j=f[j];
  }
  if (texto[k+1])==patron[j+1]))
  {
    j++;
  }
  k++;
}
// j==m => calce, j el patron estaba en el texto
```

Ejemplo:

Patron = "a a b a a a"
1 2 3 4 5 6

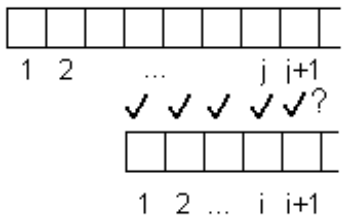
j	1	2	3	4	5	6
f(j)	0	1	0	1	2	2

Texto: "a a a a b a a b a a a b b"
j = 0 1 2
1 2
1 2 3 4 5
2 3 4 5 6 → calce!

El tiempo de ejecución de este algoritmo no es difícil de analizar, pero es necesario ser cuidadoso al hacerlo. Dado que se tienen dos ciclos anidados, se puede acotar el tiempo de ejecución por el número de veces que se ejecuta el ciclo externo (menor o igual a *n*) por el número de veces que se ejecuta el ciclo interno (menor o igual a *m*), por lo que la cota es igual a $O(mn)$, ¡que es igual a lo que demora el algoritmo de fuerza bruta!.

El análisis descrito es pesimista. Note que el número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede decrementar *j*, dado que *f(j)<j*. Pero *j* comienza desde cero y es siempre mayor o igual que cero, por lo que dicho número es menor o igual al número de veces que *j* es incrementado, el cual es menor que *n*. Por lo tanto, el tiempo total de ejecución es $O(n)$. Por otra parte, *k* nunca es decrementado, lo que implica que el algoritmo nunca se devuelve en el texto.

Queda por resolver el problema de definir la función de fracaso, *f(j)*. Esto se puede realizar inductivamente. Para empezar, *f(1)=0* por definición. Para calcular *f(j+1)* suponga que ya se tienen almacenados los valores de *f(1), f(2), ..., f(j)*. Se desea encontrar un *i+1* tal que el *(i+1)*-ésimo carácter del patrón sea igual al *(j+1)*-ésimo carácter del patrón.



Para esto se debe cumplir que *i=f(j)*. Si $b_{i+1}=b_{j+1}$, entonces *f(j+1)=i+1*. En caso contrario, se reemplaza *i* por *f(i)* y se verifica nuevamente la condición.

El algoritmo resultante es el siguiente (note que es similar al algoritmo KMP):

```
// m es largo del patron
// los indices comienzan desde 1
int[] f=new int[m];
f[1]=0;
int j=1;
int i;
while (j<m)
{
  i=f[j];
  while (i>0 && patron[i+1]!=patron[j+1])
  {
```

```
    i=f[i];
  }
  if (patron[i+1]==patron[j+1])
  {
    f[j+1]=i+1;
  }
  else
  {
    f[j+1]=0;
  }
  j++;
}
```

El tiempo de ejecución para calcular la función de fracaso puede ser acotado por los incrementos y decrementos de la variable *i*, que es $O(m)$.

Por lo tanto, el tiempo total de ejecución del algoritmo, incluyendo el preprocesamiento del patrón, es $O(n+m)$.

Algoritmo Boyer-Moore

Hasta el momento, los algoritmos de búsqueda en texto siempre comparan el patrón con el texto de izquierda a derecha. Sin embargo, suponga que la comparación ahora se realiza de derecha a izquierda: si hay una discrepancia en el último carácter del patrón y el carácter del texto no aparece en todo el patrón, entonces éste se puede deslizar *m* posiciones sin realizar niguna comparación extra. En particular, no fue necesario comparar los primeros *m-1* caracteres del texto, lo cual indica que podría realizarse una búsqueda en el texto con menos de *n* comparaciones; sin embargo, si el carácter discrepante del texto se encuentra dentro del patrón, éste podría desplazarse en un número menor de espacios.

El método descrito es la base del algoritmo Boyer-Moore, del cual se estudiarán dos variantes: Horspool y Sunday.

Boyer-Moore-Horspool (BMH)

El algoritmo BMH compara el patrón con el texto de derecha a izquierda, y se detiene cuando se encuentra una discrepancia con el texto. Cuando esto sucede, se desliza el patrón de manera que la letra del texto que estaba alineada con b_m, denominada *c*, ahora se alinie con algún b_j, con $j<m$, si dicho calce es posible, o con b_0, un carácter ficticio a la izquierda de b_1, en caso contrario (este es el mejor caso del algoritmo).

Para determinar el desplazamiento del patrón se define la *función siguiente* como:

- 0 si *c* no pertenece a los primeros *m-1* caracteres del patrón (¿Por qué no se considera el carácter b_m?).
- *j* si *c* pertenece al patrón, donde $j<m$ corresponde al mayor índice tal que $b_j==c$.

Esta función sólo depende del patrón y se puede precalcular antes de realizar la búsqueda.

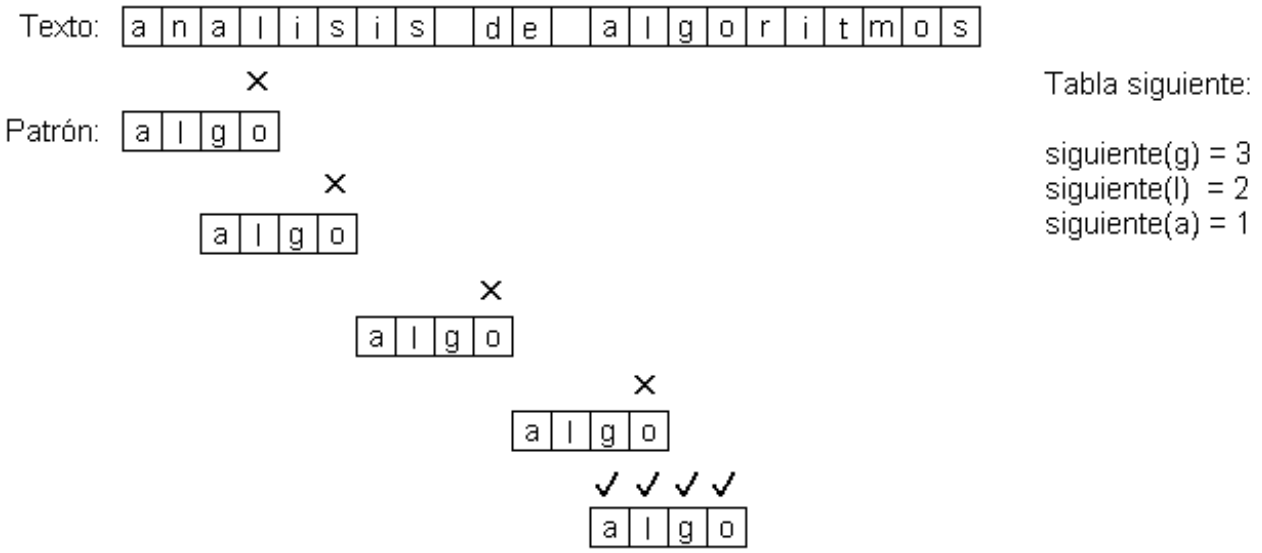
El algoritmo de búsqueda es el siguiente:

```
// m es el largo del patron
```

```
// los indices comienzan desde 1

int k=m;
int j=m;
while(k<=n && j>=1)
{
  if (texto[k-(m-j)]==patron[j])
  {
    j--;
  }
  else
  {
    k=k+(m-siguiente(a[k]));
    j=m;
  }
}
// j==0 => calce!, j>=0 => no hubo calce.
```

Ejemplo de uso del algoritmo BMH:



Se puede demostrar que el tiempo promedio que toma el algoritmo BMH es:

$$O\left(n\left(\frac{1}{m}+\frac{1}{2c}\right)\right)$$

donde *c* es el tamaño del alfabeto (*c*<<*n*). Para un alfabeto razonablemente grande, el algoritmo es $O\left(\frac{n}{m}\right)$.

En el peor caso, BMH tiene el mismo tiempo de ejecución que el algoritmo de fuerza bruta.

Boyer-Moore-Sunday (BMS)

El algoritmo BMH desliza el patrón basado en el símbolo del texto que corresponde a la posición del último carácter del patrón. Este siempre se desliza al menos una posición si se encuentra una discrepancia con el texto.

Es fácil ver que si se utiliza el carácter una posición más adelante en el texto como entrada de la función siguiente el algoritmo también funciona, pero en este caso es necesario considerar el

patrón completo al momento de calcular los valores de la función siguiente. Esta variante del algoritmo es conocida como Boyer-Moore-Sunday (BMS).

¿Es posible generalizar el argumento, es decir, se pueden utilizar caracteres más adelante en el texto como entrada de la función siguiente? La respuesta es no, dado que en ese caso puede ocurrir que se salte un calce en el texto.