ALGORITMOS DE BÚSQUEDA EN CADENAS:

SHIFT AND, SHIFT OR Y BMH

SHIFT AND , SHIFT OR

- Los algoritmos Shift-And y Shift-Or están relacionados, Shift-Or es solo una mejora en la implementación.

- Los algoritmos mantienen el conjunto de todos los prefijos de p que coinciden con un sufijo del texto leído. Usan bitparallelism para actualizar este conjunto para cada nuevo carácter de texto. El conjunto está representado por una máscara de bits D = dm,. . . , d1.

SHIFT AND:

- Preprocesamiento:
  - El algoritmo crea una tabla B que almacena una bitmasks bm . . . , b1 para cada carácter α ∈ Σ. B [α] tiene el bit [j] establecido si pj = α. El cálculo de B toma O (m + | Σ |).
- Búsqueda:
  - Inicialmente establecemos D = 0^m y para cada nuevo carácter t_{pos} actualizamos D usando la fórmula:

$$D' = \left((D \ll 1) \mid 0^{m-1}1\right) \,\&\, B[t_{pos}]$$

PSEUDOCÓDIGO SHIFT-AND

Algorithm 1 Algoritmo Shift-And

```
1: Input : T de longitud n , patrón P de longitud m
2: Output : todas las apariciones de P en T
3: Preprocesamiento ;
4: for c ∈ Σ do
5:     B[c] = 0^m
6: end for
7: for j ∈ 1...m do
8:     B[p_j] = B[p_j] | 0^{m-j}10^{j-1}
9: end for
10: Búsqueda :
11: D = 0^m
12: for pos ∈ 1...n do
13:     D = ((D << 1)|0^{m-1}1) & B[t_pos]
14:     if D & 10^{m-1} ≠ 0^m then
15:         P es ocurrencia en la posición pos - m + 1
16:     end if
17: end for
```

SHIFT-AND VS SHIFT-OR

- El algoritmo Shift-Or es solo una mejora en la implementación para evitar la operación “| 0^(m−1) 1”.
- En el algoritmo Shift-Or complementamos todas las máscaras de bits de B y usamos una máscara de bits complementada D.

PSEUDOCÓDIGO SHIFT-OR

Algorithm 2 Algoritmo Shift Or

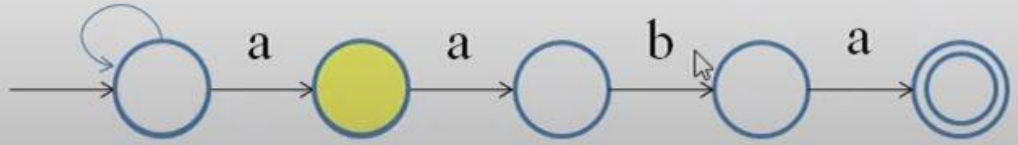
```
1: Input : T de longitud n , patrón P de longitud m
2: Output : todas las apariciones de P en T
3: Preprocesamiento ;
4: for c ∈ Σ do
5:     B[c] = ∼ 0
6: end for
7: for j ∈ 1...m do
8:     B[p_j] &= ∼ (1 << j)
9: end for
10: Búsqueda :
11: D' = ∼ 0^m
12: for pos ∈ 1...n do
13:     D' = (D << 1)|B[t_pos]
14:     if D' & 10^{m-1} = 0^m then
15:         P es ocurrencia en la posición pos - m + 1
16:     end if
17: end for
```

- Pattern: aaba

Character	b_1	b_2	b_3	b_4	bit mask ($b_4b_3b_2b_1$)
a	0	0	1	0	0100
b	1	1	0	1	1011
*	1	1	1	1	1111

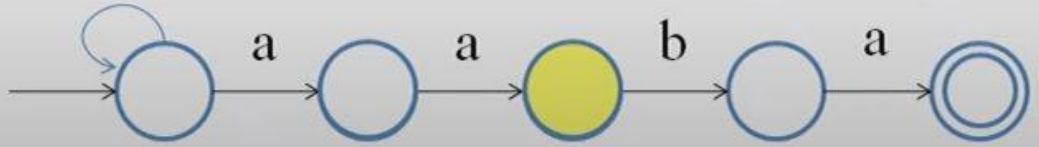
T:	a	a	b	a	a	c	a	a	d	a	a	b	a

Initial state (D)	T_j	$B[T_j]$	$D \ll 1$	$(D \ll 1) \mid B[Tj]$
1111	a	0100	1110	1110



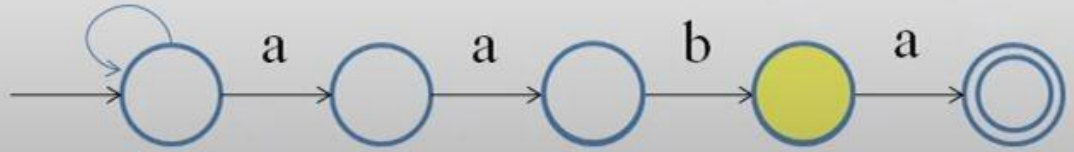
T:	a	a	b	a	a	c	a	a	d	a	a	b	a

State (D)	T_j	$B[T_j]$	$D \ll 1$	$(D \ll 1) \mid B[Tj]$
1110	a	0100	1100	1100



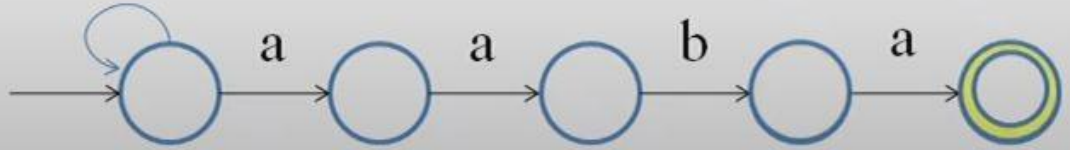
T:	a	a	b	a	a	c	a	a	d	a	a	b	a

State (D)	T_j	$B[T_j]$	$D \ll 1$	$(D \ll 1) \mid B[Tj]$
1100	b	1011	1000	1011



T:	a	a	b	a	a	c	a	a	d	a	a	b	a

State (D)	T_j	$B[T_j]$	$D \ll 1$	$(D \ll 1) \mid B[Tj]$
1011	a	0100	0110	0110



BOYER MOORE HORSPOOL

BOYER MOORE(BM)

- Fue desarrollado por Bob Boyer y Strother Moore en 1977.
- En cada alineación del patrón con el texto, los caracteres del texto debajo del patrón se examinan de derecha a izquierda.
- El desplazamiento sobre el texto se hace de izquierda a derecha.
- El desplazamiento se calcula utilizando dos heurísticas: la heurísticade coincidencia y la heurística de ocurrencia.

ALGORITMO DE BOYER MOORE HORSPOOL

- El algoritmo de Boyer Moore Horspool(BMH) este algoritmo fue publicado por Nigel Horspool en 1980.
- Este algoritmo se divide en dos fases, la fase de preprocesamiento y procesamiento.
- El algoritmo BMH busca el patrón de izquierda a derecha sin embargo el modo de recorrer ambos strings es de derecha a izquierda al igual que en el algoritmo de Boyer Moore.
- El algoritmo BMH tiene un código simple y en la práctica es mejor que el algoritmo original de Boyer-Moore.

Fase de Preprocesamiento BMH

En la fase de preprocesamiento, el algoritmo calcula a partir del patrón la tabla de cambios ***d*** definida para cada símbolo ***a*** en el alfabeto Σ.

$$d[a] = min\{s | s = m \; or (1 \leq s < m \; and \; p_{m-s} = a)\}$$

Algorithm 3 Algoritmo de preprocesamiento BMH

```
1: for a en Σ do
2:     d[a] ← m
3: end for
4: for i ← 1 hasta m − 1 do
5:     d[p_i] ← m − i
6: end for
```

FASE DE PROCESAMIENTO BMH

En esta fase se determina el algoritmo BMH como tal haciendo uso de una tabla pre-calculada para determinar la distancia de saltos en el texto.

El algoritmo ***BMH*** que incluye el escaneo del texto T.

Algorithm 4 Algoritmo BMH

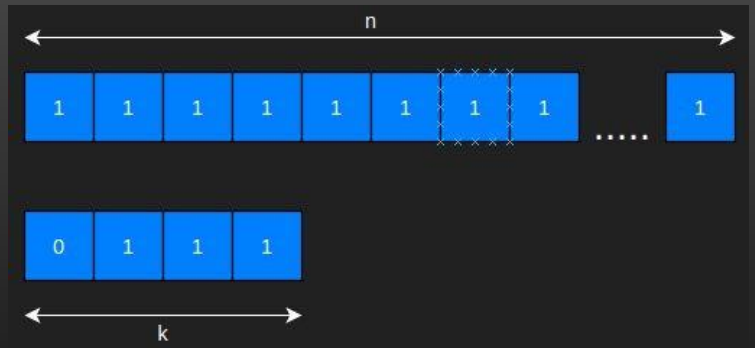
```
1: Usar algoritmo de preprocesamiento
2: j ← m                                   ▷ el patrón termina en la posición del texto j
3: while j ≤ n do
4:     h ← j                                                    ▷ h escanea el texto
5:     i ← m                                                    ▷ i escanea el patrón
6:     while i > n  &  t_h = p_i do                             ▷ nos movemos a la izquierda
7:         i ← i − 1
8:         h ← h − 1
9:     end while
10:    if i = 0 then
11:        Se encontro en la posicion j
12:    else
13:        j ← j + d[t_j]                                       ▷ nos desplazamos a la derecha
14:    end if
15: end while
```

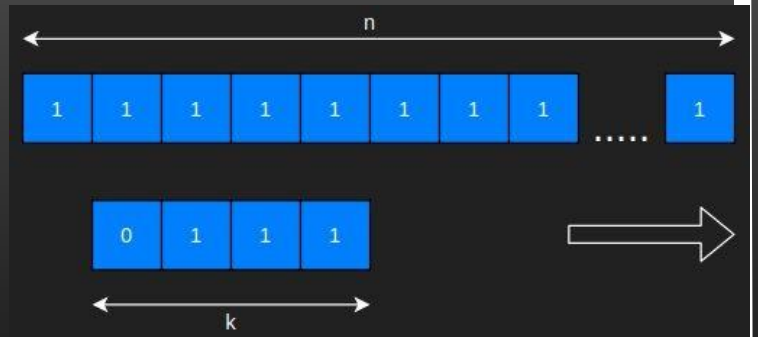
Análisis: Supongamos el siguiente caso.

Se tiene una cadena de caracteres con un tamaño ***n*** con puros ***unos*** y como patrón se tiene una cadena de caracteres que empieza con un ***cero*** y termina con una serie de ***unos*** con una longitud ***k***

ANÁLISIS: PEOR CASO ***O(N.K)***

Como se va a buscar desde el carácter más a la derecha del patrón y a partir de ese ındice un barrido a la izquierda sobre el mismo patrón, y en este caso en específico recorreremos todo el patrón hasta encontrar el ***0*** en la primera posición y teniendo los datos como el texto conformado de ***1s*** y el patrón conformado con un solo carácter diferente al texto, esto genera una tabla de saltos con un valor de ***1***, lo que significa que siempre se estará desplazando de ***1*** en ***1*** a lo largo del texto teniendo una complejidad de O(n.k)


n
1 1 1 1 1 1 1 1 1
0 1 1 1
k

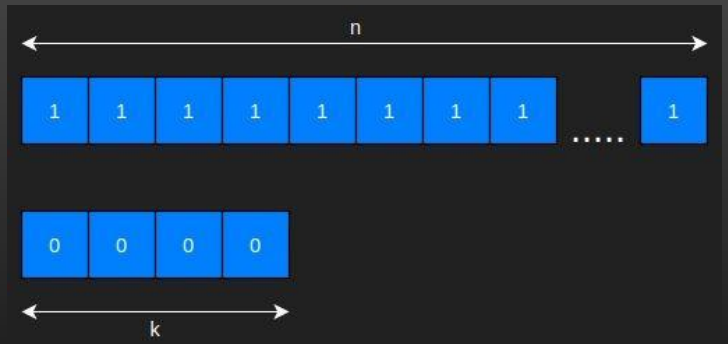

n
1 1 1 1 1 1 1 1 1
0 1 1 1
k

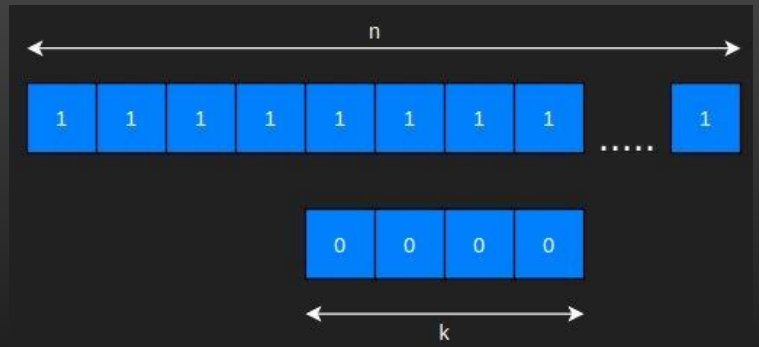

0 1
1 1
? 1

ANÁLISIS: PEOR CASO ***O(N/K)***

Para este caso se modifica el valor del ***patrón*** del ejemplo inicial y así obtener el mejor de los casos, el valor actual del ***patrón*** ya no estará conformada de un primer caracter diferente al texto si no que la totalidad de sus caracteres serán distintos al ***texto***, entonces, supongamos que se tiene un ***patrón*** conformado de ***0s*** con una longitud ***k*** y un ***texto*** conformado de ***1s*** de tamaño, tal cual el texto del ejemplo inicial.

Entonces siguiendo el algoritmo de preprocesamiento BMH se arma una tabla de saltos con el único valor de 4, forzando el desplazamiento en saltos del tamaño del patrón en el texto dando una complejidad de ***o(n/k)***

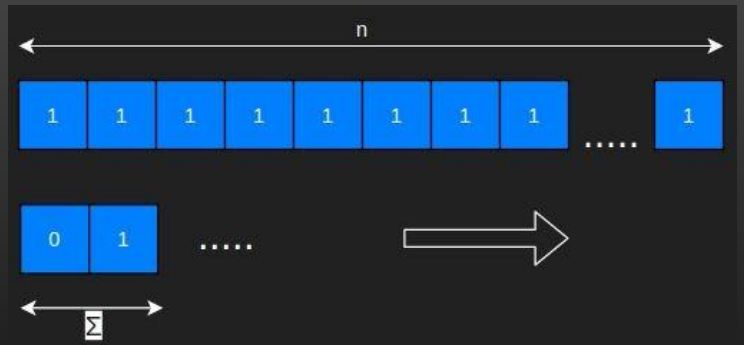

n
1 1 1 1 1 1 1 1 1
0 0 0 0
k


n
1 1 1 1 1 1 1 1 1
0 0 0 0
k

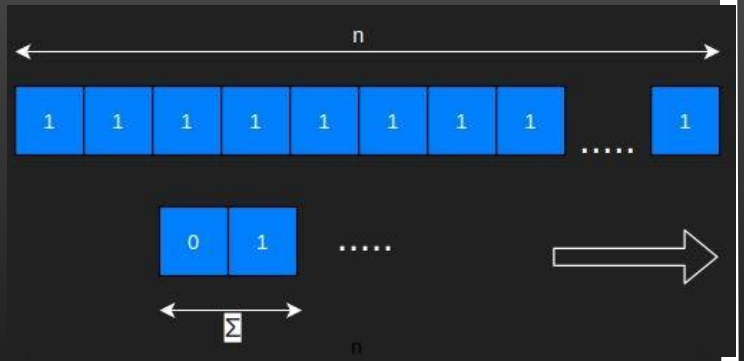

0 4
? 4

ANÁLISIS: CASO PROMEDIO *Θ(N/Σ)*

Un caso más realista es este caso, al analizar este caso se toma en cuenta los datos del tamaño del texto y el alfabeto que se está usando, para este caso 0 y 1 llegando a una complejidad de Θ(n/Σ), donde sigma es el tamaño del alfabeto usado.


n
1 1 1 1 1 1 1 1 1
0 1 .....
Σ


0 Σ
1 Σ
? Σ


n
1 1 1 1 1 1 1 1 1
0 1 .....
Σ
n

PROBLEMAS PROPUESTOS

COINCIDENCIA DE CADENAS

Aporte

La entrada consta de varios casos de prueba. Cada caso de prueba consta de dos líneas, primero un patrón no vacío y luego un texto no vacío. La entrada finaliza al final del archivo. El archivo de entrada no superará los 5 Mb.

Producción

Para cada caso de prueba, genere una línea que contenga las posiciones de todas las ocurrencias de patrón en el texto, de la primera a la última, separadas por un solo espacio.

Sample Input 1

```
p
Popup
helo
Hello there!
peek a boo
you speek a bootiful language
anas
bananananaspaj
```

Sample Output 1

```
2 4

5
7
```

```cpp
#include <bits/stdc++.h>
#define SIGMA 256

using namespace std;

int* tablaBMHS(string patron) {
  int m = patron.size();
  int maxChar = SIGMA;
  int* bmhsTable = new int[maxChar];
  for (int i = 0; i < maxChar; i++)
    bmhsTable[i] = m + 1;

  for (int i = 0; i < m; i++)
    bmhsTable[patron.at(i)] = m - i;

  return bmhsTable;
}

void BMHS(string cadena, string patron) {
  int n = cadena.size()-1;
  int m = patron.size();

  // preprocesamiento
  int* bmhsTable = tablaBMHS(patron);

  int i = m - 1;
  int contador = 0;
  while (i < n) {
    int k = i;
    int j = m - 1;
    while (j >= 0 && (cadena.at(k) == patron.at(j))) {
      j--;
      k--;
    }
    if (j < 0) {
      cout << (k + 1) << " ";
      contador++;
    }
    i = i + bmhsTable[cadena.at(i + 1)];
  }
  cout << "\n";
}
```

Sample Input 1

```
p
Popup
helo
Hello there!
peek a boo
you speek a bootiful language
anas
banananananaspaj
```

Sample Output 1

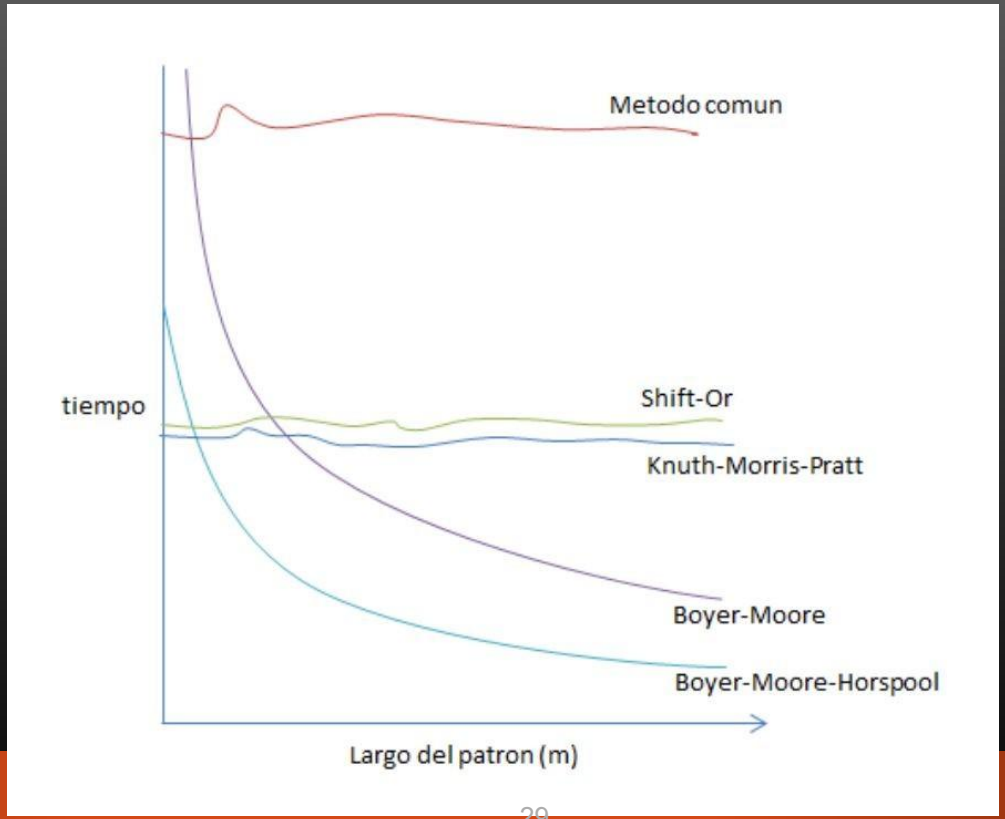
```
2 4

5
7
```

COMPLEJIDAD

```cpp
#include <bits/stdc++.h>
#define SIGMA 256

using namespace std;

int* tablaBMHS(string patron) {
  int m = patron.size();
  int maxChar = SIGMA;
  int* bmhsTable = new int[maxChar];
  for (int i = 0; i < maxChar; i++)
    bmhsTable[i] = m + 1;

  for (int i = 0; i < m; i++)
    bmhsTable[patron.at(i)] = m - i;

  return bmhsTable;
}

void BMHS(string cadena, string patron) {
  int n = cadena.size()-1;
  int m = patron.size();

  // preprocesamiento
  int* bmhsTable = tablaBMHS(patron);

  int i = m - 1;
  int contador = 0;
  while (i < n) {
    int k = i;
    int j = m - 1;
    while (j >= 0 && (cadena.at(k) == patron.at(j))) {
      j--;
      k--;
    }
    if (j < 0) {
      cout << (k + 1) << " ";
      contador++;
    }
    i = i + bmhsTable[cadena.at(i + 1)];
  }
  cout << "\n";
}
```

CONCLUSIONES

- Los algoritmo Shift And, Shift Or, se desempeñan mejor con un patrón corto.
- El algoritmo de Booyer Moore Horspol es más fácil de implementar que el algoritmo original Booyer Moore.
- El algoritmo de Boyer Moore Horspool(BMH) a diferencia de los algoritmos Shift And, Shift Or, tiene un mejor rendimiento si el patrón el cual se busca tiene más caracteres.
- El algoritmo de Boyer Moore Horspool(BMH) se desempeña mejor con respecto a la longitud del patrón, como se ve en la siguiente gráfico.


Metodo comun
tiempo
Shift-Or
Knuth-Morris-Pratt
Boyer-Moore
Boyer-Moore-Horspool
Largo del patron (m)

REFERENCIAS

Hancarville, P. and Niemoeller, A., (1949). The Private Lives Of The Twelve Caesars. Girard, Kan.

Wobst, R. (2001). Cryptology Unlocked. Wiley. p. 19

Menezes, Alfred J.; Oorschot, Paul C. van; Vanstone, Scott A. (2001). Handbook of Applied Cryptography (Fifth ed.). p.251

J. Smith. (1971) The design of Lucifer: a cryptographic device for data communications. IBM Research Report RC 3326, IBM T.J. Watson Research Center, Yorktown Heights, New York, USA.

M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. SIAM Journal on Computing, 17(2), pp. 373–386, 1988.

R. Boyer and S. Moore: A fast string searching algorithm.

D. Knuth, J. Morris and V. Pratt: Fast pattern matching in strings.
Wikipedia : https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm.

González Saavedra, Tomás Ignacio: Análisis Suavizado del Algoritmo Boyer-Moore-Horspool