

# String Matching

Algoritmos y Estructuras de Datos I

1

## Strings

- ▶ Llamamos un **string** a una secuencia de **Char**.
- ▶ Los strings no difieren de las secuencias sobre otros tipos, dado que habitualmente no se utilizan operaciones particulares de los **Chars**.
- ▶ Los strings aparecen con mucha frecuencia en diversas aplicaciones.
  1. Palabras, oraciones y textos.
  2. Nombres de usuario y claves de acceso.
  3. Secuencias de ADN.
  4. Código fuente!
  5. ...
- ▶ El estudio de **algoritmos sobre strings** es un tema muy importante.

2

## Ejemplo

a	b	a	b	a	c	a
---	---	---	---	---	---	---

a	b	b	x	c	a
---	---	---	---	---	---

3

## Búsqueda de un patrón en un texto

- ▶ **Problema:** Dado un string  $t$  (texto) y un string  $p$  (patrón), queremos saber si  $p$  se encuentra dentro de  $t$ .
- ▶ **Notación:** La función  $subseq(t, d, h)$  denota al substring de  $t$  entre  $d$  y  $h - 1$  (inclusive).
- ▶ 

```
proc contiene(in  $t, p : seq\langle Char \rangle$ , out  $result : Bool$ ){  
  Pre {  $True$  }  
  Post {  $result = true \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i \leq |t| - |p|$   
     $\wedge_L subseq(t, i, i + |p|) = p)$  }  
}
```
- ▶ ¿Cómo resolvemos este problema?

4

## Función Auxiliar matches

- Implementemos una función auxiliar con la siguiente especificación:
- `proc matches(in s : seq<Char>, in i : ℤ,  
in r : seq<Char>, out result : Bool){`  
Pre  $\{0 \leq i < |s| - |r| \wedge |r| \leq |s|\}$   
Post  $\{result = true \leftrightarrow subseq(s, i, i + |r|) = r\}$   
}

5

## Función Auxiliar matches

```
bool matches(string &s, int i, string &r) {  
    bool result = true;  
    for (int k = 0; k < r.size() ; k++) {  
        if (s[i+k] != r[k]) {  
            result = false;  
        }  
    }  
    return result;  
}
```

¿Se puede hacer que sea más eficiente?

6

## Función Auxiliar matches

```
bool matches(string &s, int i, string &r) {  
    int k = 0;  
    while (k < r.size() && s[i+k] == r[k]) {  
        k++;  
    }  
    return k == r.size();  
}
```

Este programa se interrumpe tan pronto como detecta una desigualdad.

7

## Función Auxiliar matches

```
bool matches(string &s, int i, string &r) {  
  
    for (k = 0; k < r.size() && s[i+k] == r[k]; k++) {  
        // skip  
    }  
    return k == r.size();  
}
```

Este programa se interrumpe tan pronto como detecta una desigualdad.

¿Cuál es la complejidad?

En peor caso, el for se ejecuta  $|r| + 1$  veces y las demás son operaciones elementales. Por lo tanto  $T_{matches}(s, i, r) \in O(|r|)$

8

## Búsqueda de un patrón en un texto

- **Algoritmo sencillo:** Recorrer las posiciones  $i$  de  $t$  donde podría empezar  $p$  y verificar si  $subseq(t, i, i + |p|) = p$ .

```
bool contiene(string &t, string &p) {  
    for (int i=0; i<=t.size()-p.size(); i++) {  
        && !matches(t,i,p); i++) {  
            // skip  
        }  
    }  
    return i<=t.size()-p.size();  
}
```

- `matches` es la función auxiliar definida anteriormente.

9

## Búsqueda de un patrón en un texto

- ¿Cuál es el tiempo de ejecución en peor caso de para la función contiene?

```
bool contiene(string &t, string &p) {  
    for (int i=0; i<=t.size()-p.size(); i++) {  
        && !matches(t,i,p); i++) { // O(|p|)  
            // skip  
        }  
    }  
    return i<=t.size()-p.size(); // O(1)  
}
```

- El for se ejecuta  $|t| - |p|$  veces en peor caso.
- La comparación `matches(t,i,p)` requiere realizar  $|p|$  comparaciones entre chars.
- $T_{contiene}(t, p) \in O(|p|) * O(|t|) = O(|p| * |t|)$

10

## Algoritmo de Knuth, Morris y Pratt

- En 1977, Donald Knuth, James Morris y Vaughan Pratt propusieron un algoritmo más eficiente.
- **Idea:** Tratar de no volver a inspeccionar **todo** el patrón cada vez que avanzamos en el texto.
- Mantenemos dos **índices**  $l$  (left) y  $r$  (right) a la secuencia, con el siguiente invariante:
  1.  $0 \leq l \leq r \leq |t|$
  2.  $subseq(t, l, r) = subseq(p, 0, r - l)$
  3. No hay apariciones de  $p$  en  $subseq(t, 0, r)$ .

11

## Algoritmo de Knuth, Morris y Pratt

- Planteamos el siguiente esquema para el algoritmo.

```
bool contiene_kmp(string &t, string &p) {  
    int l = 0, r = 0;  
    while( r < t.size() ) {  
        // Aumentar l o r  
        // Verificar si encontramos p  
    }  
    return // resultado  
}
```

- ¿Cómo aumentamos  $l$  o  $r$  **preservando** el invariante?

12

## Algoritmo de Knuth, Morris y Pratt

- ▶ Si  $r - l = |p|$ , entonces encontramos  $p$  en  $t$ .
- ▶ Si  $r - l < |p|$ , consideramos los siguientes casos:
  1. Si  $t[r] = p[r - l]$ , entonces encontramos una nueva coincidencia, y entonces incrementamos  $r$  para reflejar esta nueva situación.
  2. Si  $t[r] \neq p[r - l]$  y  $l = r$ , entonces no tenemos un **prefijo** de  $p$  en el texto, y pasamos al siguiente elemento de la secuencia avanzando  $l$  y  $r$ .
  3. Si  $t[r] \neq p[r - l]$  y  $l < r$ , entonces debemos avanzar  $l$ .  
¿Cuánto avanzamos  $l$  en este caso? ¡Tanto como podamos!  
(más sobre este punto a continuación)

13

## Algoritmo (parcial) de Knuth, Morris y Pratt

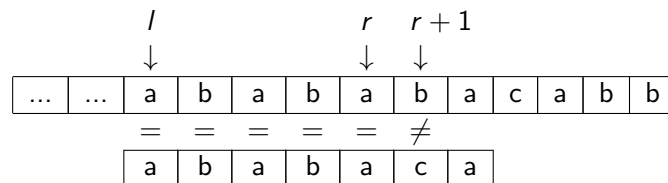
```
bool contiene_kmp(string &t, string &p) {
    int l = 0, r = 0;
    while( r < t.size() && r-l < p.size()) {
        if( t[r] == p[r-l] ){
            r++;
        } else if( l == r ) {
            r++;
            l++;
        } else {
            l = // avanzar l
        }
    }
    return r-l == p.size();
}
```

14

## Algoritmo de Knuth, Morris y Pratt

- ▶ Si  $t[r] \neq p[r - l]$  y  $l < r$ , ¿cuánto podemos avanzar  $l$ ?
- ▶ El invariante implica que  $subseq(t, l, r) = subseq(p, 0, r - l)$ , pero esta condición dice que  $subseq(t, l, r + 1) \neq subseq(p, 0, r + 1 - l)$ .

- ▶ Ejemplo:



- ▶ ¿Hasta donde puedo avanzar  $l$ ?

15

## Bifijos: Prefijo y Sufijo simultáneamente

- ▶ **Definición:** Una cadena de caracteres  $b$  es un *bifijo* de  $s$  si  $b \neq s$ ,  $b$  es un prefijo de  $s$  y  $b$  es un sufijo de  $s$ .
- ▶ Ejemplos:

$s$	bifijos
a	$\langle \rangle$
ab	$\langle \rangle$
aba	$\langle \rangle, a$
abab	$\langle \rangle, ab$
ababc	$\langle \rangle$
aaaa	$\langle \rangle, a, aa, aaa, aaa$
abc	$\langle \rangle$
ababaca	$\langle \rangle, a$

- ▶ **Observación:** Sea una cadena  $s$ , su máximo bifijo es **único**.

16

## KMP: Función $\pi$

- **Definición:** Sea  $\pi(i)$  la longitud del **máximo** bifijo de  $subseq(p, 0, i)$
- Por ejemplo, sea  $p=abbabbaa$ :

$i$	$subseq(p, 0, i)$	Máx. bifijo	$\pi(i)$
1	a	$\langle \rangle$	0
2	ab	$\langle \rangle$	0
3	abb	$\langle \rangle$	0
4	abba	a	1
5	abbab	ab	2
6	abbabb	abb	3
7	abbabba	abba	4
8	abbabbaa	a	1

17

## KMP: Función $\pi$

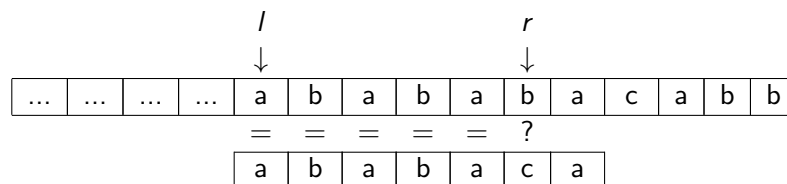
- **Definición:** Sea  $\pi(i)$  la longitud del **máximo** bifijo de  $subseq(p, 0, i)$
- Otro ejemplo, sea  $p=ababaca$ :

$i$	$subseq(p, 0, i)$	Máx. bifijo	$\pi(i)$
1	a	$\langle \rangle$	0
2	ab	$\langle \rangle$	0
3	aba	a	1
4	abab	ab	2
5	ababa	aba	3
6	ababac	$\langle \rangle$	0
7	ababaca	a	1

18

## Algoritmo de Knuth, Morris y Pratt

- **Ejemplo:** Supongamos que ...

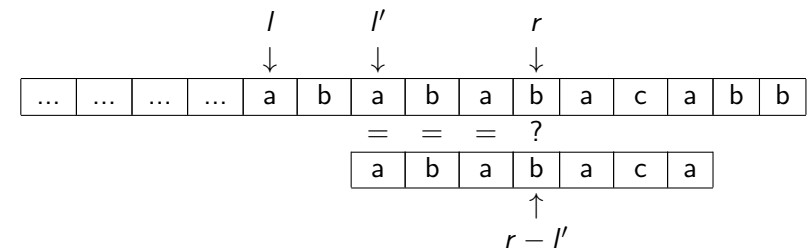


- En este caso, podemos avanzar  $l$  hasta la posición **ababa** ( $\pi(r-l) = \pi(5) = 3$ ), dado que no tendremos coincidencias en las posiciones anteriores.
- Por lo tanto, en este caso fijamos  $l' = r - \pi(r-l)$ .

19

## Algoritmo de Knuth, Morris y Pratt

- **Ejemplo:** Supongamos que ...



- En este caso, podemos avanzar  $l$  hasta la posición **ababa** ( $\pi(r-l) = \pi(5) = 3$ ), dado que no tendremos coincidencias en las posiciones anteriores.
- Por lo tanto, en este caso fijamos  $l' = r - \pi(r-l)$ .

20

## Algoritmo de Knuth, Morris y Pratt

- Podemos calcular  $\pi(i)$  usando una función auxiliar.
- Supongamos que ya tenemos una función auxiliar `calcular_pi(p)` que retorna una secuencia de enteros con los valores de  $\pi$ ,
- Asumamos que su tiempo de ejecución de peor caso  $\in O(|p|)$
- Implementemos ahora `contieneKMP`

21

## Algoritmo (parcial) de Knuth, Morris y Pratt

```
bool contiene_kmp(string &t, string &p) {
    int l = 0, r = 0;
    while( r < t.size() && r-l < p.size()) {
        if( t[r] == p[r-l] ){
            r++;
        } else if( l == r ) {
            r++;
            l++;
        } else {
            vector<int> pi = calcular_pi(p);
            l = r - pi[r-l];
        }
    }
    return r-l == p.size();
}
```

23

## Algoritmo de Knuth, Morris y Pratt

Recordemos el invariante para el algoritmo KMP:

1.  $0 \leq l \leq r \leq |t|$
  2.  $subseq(t, l, r) = subseq(p, 0, r - l)$
  3. No hay apariciones de  $p$  en  $subseq(t, 0, r)$ .
- ¿Se cumplen los tres puntos del teorema del invariante?
    1. El invariante vale con  $l = r = 0$ .
    2. Cada caso del `if...` preserva el invariante.
    3. Al finalizar el ciclo, el invariante permite retornar el valor correcto.
  - ¿Cómo es una función variante para este ciclo?
    - Notar que en cada iteración se aumenta  $l$  o  $r$  (o ambas) en al menos una unidad.
    - Entonces, una función variante puede ser:
$$fv = (|t| - l) + (|t| - r) = 2 * |t| - l - r$$
  - Es fácil ver que se cumplen los dos puntos del teorema de terminación del ciclo, y por lo tanto el ciclo termina.

24

## Algoritmo de Knuth, Morris y Pratt

- Para completar el algoritmo debemos calcular  $\pi(i)$ .
- Para este cálculo, recorreremos  $p$  con dos índices  $i$  y  $j$ , con el siguiente invariante:
  1.  $0 \leq i < j \leq |p|$
  2.  $pi[k] = \pi(k+1)$  para  $k = 0, \dots, j-1$  (vector empieza en 0)
  3.  $i$  es la longitud del máximo bifijo para  $subseq(p, 0, j)$ .
  4.  $0 \leq \pi(j) \leq i+1$

25

## Algoritmo de Knuth, Morris y Pratt

```
vector<int> calcular_pi(string &p) {  
    vector<int> pi(p.size(),0); // inicializado en 0  
    int i = 0;  
    for(int j=1; j < p.size(); j++) {  
        // Si no coincide busco bifijo mas chico  
        while(i>0 && p[i] != p[j])  
            i = pi[i-1];  
  
        // Si coincide, aumento tamaño bifijo  
        if( p[i] == p[j] )  
            i++;  
  
        pi[j] = i;  
    }  
    return pi;  
}
```

Veamos como funciona `calcular_pi()` con el patrón  
`<a, b, a, b, a, c, a>`

26

## Algoritmo de Knuth, Morris y Pratt

- ¡Es importante observar que sin el invariante, es muy difícil entender este algoritmo!
- ¿Cómo es una función variante adecuada para el ciclo?
  1. ¿Para el loop interno?  $fv = i$
  2. ¿y para el externo?  $fv = |p| - j$ .
- ¿Y el tiempo de ejecución de peor caso?
  1. siempre se incrementa  $j$
  2.  $i$  disminuye a lo sumo  $|p|$ -veces sumando todas las  $j$  iteraciones!
- Entonces, el tiempo de ejecución de peor caso de `calcular_pi`  $\in O(|p| + |p|) = O(|p|)$

27

## Algoritmo (completo) de Knuth, Morris y Pratt

```
bool contiene_kmp(string &t, string &p) {  
    int l = 0, r = 0;  
    vector<int> pi = calcular_pi(p);  
    while( r < t.size() && r-l < p.size() ) {  
        if( t[r] == p[r-l] ) {  
            r++;  
        } else if( l == r ) {  
            r++;  
            l++;  
        } else {  
            l = r - pi[r-l];  
        }  
    }  
    return r-l == p.size();  
}
```

Tiene tiempo de ejecución de peor caso  $\in O(|t| + |p|)$

29

## Algoritmo de Knuth, Morris y Pratt

¿Es realmente mejor la eficiencia de KMP en comparación con la solución trivial?



Veamos como funciona cada algoritmo en la computadora

<http://whocouldthat.be/visualizing-string-matching/>

30

## Algoritmo de Knuth, Morris y Pratt

- ▶ ¿Es realmente mejor la eficiencia de KMP en comparación con la solución trivial?
  - ▶ El algoritmo naïve tiene tiempo de ejecución de peor caso  $\in O(|t| * |p|)$
  - ▶ El algoritmo KMP tiene tiempo de ejecución de peor caso  $\in O(|t| + |p|)$
- ▶ Por lo tanto, el tiempo de ejecución de peor caso del algoritmo KMP crece asintóticamente mas despacio que el tiempo de ejecución de peor caso del algoritmo naïve.
- ▶ Existen más algoritmos de búsqueda de strings (o string matching):
  - ▶ Rabin-Karp (1987)
  - ▶ Boyer-Moore (1977)
  - ▶ Aho-Corasick (1975)

31

## Bibliografía

- ▶ David Gries - The Science of Programming
  - ▶ Chapter 16 - Developing Invariants (Linear Search, Binary Search)
- ▶ Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein- Introduction to Algorithms, 3rd edition
  - ▶ Chapter 32.1 The naive string-matching algorithm
  - ▶ Chapter 32.4 The Knuth-Morris-Pratt algorithm

32