

String Matching

Algoritmos y Estructuras de Datos I

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.

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 $subseq(t, d, h)$ es el al substring de d entre i y $h - 1$ (inclusive). Lo abreviamos como $t[d, h)$
- ▶

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

Función Auxiliar matches

- ▶ Implementemos una función auxiliar con la siguiente especificación:
- ▶

```
proc matches(in s : seq<Char>, in i :  $\mathbb{Z}$ ,  
  in r : seq<Char>, out result : Bool){  
  Pre {  $0 \leq i < |s| \wedge |r| \leq |s|$  }  
  Post { result = true  $\leftrightarrow s[i, i + |r|) = r$  }  
}
```

Función Auxiliar matches

```
1 bool matches(string &s, int i, string &r) {  
2     bool result = true;  
3     for (int k = 0; k < r.size(); k++) {  
4         if (s[i+k] != r[k]) {  
5             result = false;  
6         }  
7     }  
8     return result;  
9 }
```

¿Se puede hacer que sea más eficiente (ie: más rápido)?

Función Auxiliar matches

```
1 bool matches(string &s, int i, string &r) {  
2     int k = 0;  
3     while (k < r.size() && s[i+k] == r[k]) {  
4         k++;  
5     }  
6     return k == r.size();  
7 }
```

Este programa se interrumpe tan pronto como detecta una desigualdad.

Función Auxiliar matches

```
1 bool matches(string &s, int i, string &r) {  
2  
3     for (k = 0; k < r.size() && s[i+k] == r[k]; k++) {  
4         // skip  
5     }  
6     return k == r.size();  
7 }
```

Este programa se interrumpe tan pronto como detecta una desigualdad.

Búsqueda de un patrón en un texto

► **Algoritmo sencillo:** Recorrer todas las posiciones i de t , y para cada una verificar si $t[i, i + |p|) = p$.

```
►  
1 bool contiene(string &t, string &p) {  
2  
3     for (int i = 0; i + p.size() < t.size() && matches(t,i,p); i++) {  
4         // skip  
5     }  
6     return i + p.size() < t.size() && matches(t,i,p);  
7 }
```

► matches es una función auxiliar definida anteriormente.

Búsqueda de un patrón en un texto

- ▶ ¿Es **eficiente** este algoritmo?
- ▶ El ciclo principal realiza $|t| - |p|$ iteraciones. Sin embargo, la comparación de los substrings de t puede ser costosa si p es grande
 1. La comparación `matches(t, i, p)` requiere realizar $|p|$ comparaciones entre chars.
 2. Por cada iteración del ciclo "for", se realizan $|p|$ de estas comparaciones.
 3. En por caso, realizamos $(|t| - |p|) * |p|$ iteraciones.
- ▶ Aunque el algoritmo es eficiente si $|p|$ se aproxima a $|t|$.

Algoritmo de Knuth, Morris y Pratt

- ▶ En 1977, Donald Knuth, James Morris y Vaughan Pratt propusieron un algoritmo más eficiente.
- ▶ **Idea:** Si $t[i, i + |p|) = p$, entonces quizás podemos aprovechar parte de las coincidencias entre $[i, i + |p|)$ y p para continuar la búsqueda.
- ▶ Mantenemos dos **índices** l y r a la secuencia, con el siguiente invariante:
 1. $0 \leq r - l \leq |t|$
 2. $t[l, r) = p[0, r - l)$
 3. No hay apariciones de p en $t[0, r)$.

Algoritmo de Knuth, Morris y Pratt

- ▶ Planteamos el siguiente esquema para el algoritmo.

```
1 bool contiene_kmp(string &t, string &p) {  
2     int l = 0, r = 0;  
3     while( r < t.size() ) {  
4         // Aumentar l o r  
5         // Verificar si encontramos p  
6     }  
7     return result;  
8 }
```

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

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)

Algoritmo (parcial) de Knuth, Morris y Pratt

```

1  bool contiene_kmp(string &t, string &p) {
2      int l = 0, r = 0;
3      while( r < t.size() && r-l < p.size()) {
4          if( t[r] == p[r-l] ){
5              r++;
6          } else if( l == r ) {
7              r++;
8              l++;
9          } else {
10             l = // avanzar l
11         }
12     }
13     return r-l == p.size();
14 }

```

Algoritmo de Knuth, Morris y Pratt

- ▶ ¿Cuánto podemos avanzar l en el caso que $t[r] \neq p[r-l]$ y $l < r$?
- ▶ El invariante implica que $t[l, r) = p[0, r-l)$, pero esta condición dice que $t[l, r+1) \neq p[0, r-l+1)$.
- ▶ Ejemplo:

					l					r									
					↓					↓									
...	a	b	a	b	a	a	b	c	b	a	b					
				=	=	=	=	=	≠										
				a	b	a	b	a	c	a									

- ▶ ¿Hasta donde puedo avanzar l ?

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**.

KMP: Función π

- ▶ **Definición:** Sea $\pi(i)$ la longitud del **máximo** bifijo de $p[0, i+1)$
- ▶ Por ejemplo, sea $p = \text{abbabbaa}$:

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

Algoritmo de Knuth, Morris y Pratt

- ▶ Para completar el algoritmo debemos calcular $\pi(i)$.
- ▶ Podemos implementar una función auxiliar, pero una mejor idea es **precalcular** estos valores y guardarlos en un vector (¿por qué?).
- ▶ Para este precálculo, recorreremos p con dos índices i y j , con el siguiente invariante:
 1. $0 \leq j \leq |p|$
 2. $\pi(k) = \pi(k)$ para $k = 0, \dots, j-1$.
 3. i es la longitud de un bifijo de $p[0, j+1)$.

Algoritmo de Knuth, Morris y Pratt

```
1 vector<int> precalcular_pi(string &p) {  
2     int i = 0, j = 1;  
3     vector<int> pi(p.size()); // inicializado en 0  
4     pi[0] = 0; // valor de pi para 0  
5     while( j < p.size()) {  
6         if( p[i] == p[j] ) {  
7             pi[j] = i+1;  
8             i++;  
9             j++;  
10        } else if( i > 0 ) {  
11            i = pi[i-1];  
12        } else {  
13            pi[j] = 0;  
14            j++;  
15        }  
16    }  
17    return pi;  
18 }
```

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. En la primera rama, se incrementan i y j .
 2. En la segunda rama, se **disminuye** el valor de i .
 3. En la tercera rama, se incrementa j .
- ▶ Luego, en cada iteración se incrementa $2j - i$.
- ▶ Además, $2j - i \leq 2 \times |p|$, y entonces una función variante puede ser **$fv = 2 \times |p| - (2j - i)$** .

Algoritmo (completo) de Knuth, Morris y Pratt

```
1 bool contiene_kmp(string &t, string &p) {  
2     int l = 0, r = 0;  
3     vector<int> pi = precalcular_pi(p);  
4     while( r < t.size() && r-l < p.size()) {  
5         if( t[r] == p[r-l] ) {  
6             r++;  
7         } else if( l == r ) {  
8             r++;  
9             l++;  
10        } else {  
11            l = r - pi[r-l-1];  
12        }  
13    }  
14    return r-l == p.size();  
15 }
```

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/>

Algoritmo de Knuth, Morris y Pratt

- ▶ ¿Es realmente mejor la eficiencia de KMP en comparación con la solución trivial?
 - ▶ El algoritmo naïve realiza, en peor caso, $|t| * |p|$ iteraciones.
 - ▶ El algoritmo kmp realiza, en peor caso, $|t| + |p|$ iteraciones
- ▶ Por lo tanto, comparando sus peores casos, el algoritmo KMP es más eficiente (menos iteraciones) que el algoritmo naïve.
- ▶ Existen más algoritmos de búsqueda de strings (o string matching):
 - ▶ Rabin-Karp (1987)
 - ▶ Boyer-Moore (1977)
 - ▶ Aho-Corasick (>1977)

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