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 **Char**s.
- 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 $t, p : seq\langle Char\rangle$, out $result$: Bool){

  Pre $\{True\}$
  Post $\{result = true \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |t| - |p|$
  $\wedge_L\ t[i, i + |p|) = p)\}$
  }
- ¿Cómo resolvemos este problema?

Función Auxiliar `iguales`

- Implementemos una función auxiliar con la siguiente especificación:
- proc $iguales$(in $s : seq\langle Char\rangle$, in $i : \mathbb{Z}$,
  in $r : seq\langle Char\rangle$, in $j : \mathbb{Z}$, in ,
  $len : \mathbb{Z}$, out $result$: Bool){

  Pre $\{enRango(i, s) \wedge enRango(i + len - 1, s)$
  $\wedge enRango(j, r) \wedge enRango(j + len - 1, r)\}$
  Post $\{resut = true \leftrightarrow$
  $(\forall k : \mathbb{Z})(0 \leq k < len \rightarrow_L s[i + k] = r[j + k])\}$
  }

Función Auxiliar `iguales`

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

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

Función Auxiliar `iguales`

```
bool iguales(string &s, int i, string &r, int j, int len) {
    int k = 0;
    while (k < len && s[i+k]==r[j+k]) {
        k++;
    }
    return k == len;
}
```

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 $t[i, i + |p|) = p$.
-
```
bool contiene(string &t, string &p) {
  int i = 0;
  while ( i + p.size() < t.size() && iguales(t,i,p,0,p.size())) {
    i++;
  }
  return i + p.size() < t.size() && iguales(t,i,p,0,p.size());
}
```
- `iguales` 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 `iguales(t,i,p,0,p.size())` 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.
-
```
bool contiene_kmp(string &t, string &p) {
  int l = 0, r = 0;
  bool result = false;
  while( r < t.size() ) {
    // Aumentar l o r
    // Verificar si encontramos p
  }
  return result;
}
```
- ¿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

```
bool contiene_kmp(string &t, string &p) {
  int l = 0, r = 0;
  bool result = false;
  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();
}
```

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					
				$\downarrow$					$\downarrow$					
...	...	...	...	a	b	a	b	a	a	b	c	b	a	b
				=	=	=	=	=	$\neq$					
				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	$\langle\rangle$
abababaca	$\langle\rangle$,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=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

KMP: Función π

- **Definición:** Sea $\pi(i)$ la longitud del máximo bifijo de $p[0, i+1)$
- Otro ejemplo, sea p=abababaca:

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

Algoritmo de Knuth, Morris y Pratt

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

				l		l'			r					
				$\downarrow$		$\downarrow$			$\downarrow$					
...	...	...	...	a	b	a	b	a	a	b	c	b	a	b
				$=$	$=$	$=$	$=$	$=$	$\neq$					
				a	b	a	b	a	c	a				

- En este caso, podemos avanzar l hasta la posición *ababa*, dado que no tendremos coincidencias en las posiciones anteriores.
- Por lo tanto, en este caso fijamos $l' = r - \pi(r - l - 1)$.

Algoritmo (parcial) de Knuth, Morris y Pratt

```
bool contiene_kmp(string &t, string &p) {
  int l = 0, r = 0;
  bool result = false;
  while( r < t.size() && r−l < p.size()) {
    if( t[r] == p[r−l] ){
      r++;
    } else if( l == r ) {
      r++;
      l++;
    } else {
      l = r − calcular_pi(r−l−1);
    }
  }
  return r−l == p.size();
}
```

Algoritmo de Knuth, Morris y Pratt

- ¿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.

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, recorremos p con dos índices i y j, con el siguiente invariante:
  1. $0 \leq j \leq |p|$
  2. $pi(k) = \pi(k)$ para $k = 0, \ldots, j - 1$.
  3. i es la longitud de un bifijo de $p[0, j + 1)$.

Algoritmo de Knuth, Morris y Pratt

```
vector<int> precalcular_pi(string &p) {
  int i = 0, j = 1;
  vector<int> pi(p.size()); // inicializado en 0
  pi[0] = 0; // valor de pi para 0
  while( j < p.size()) {
    if( p[i] == p[j] ) {
      pi[j] = i+1;
      i++;
      j++;
    } else if( i > 0 ) {
      i = pi[i-1];
    } else {
      pi[j] = 0;
      j++;
    }
  }
  return pi;
}
```

Algoritmo de Knuth, Morris y Pratt

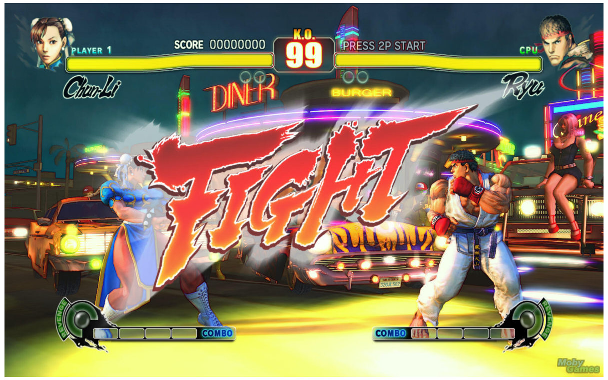
- ¡Es importante observar que sin el invariante, es muy difícil entender este algoritmo!
- Cómo es una función variante adecuada para el ciclo?
  1. En la primera rama, se incrementan i y j.
  2. En la segunda rama, se disminuye el valor de i.
  3. En la tercera rama, se incrementa j.
- Luego, en cada 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

```
bool contiene_kmp(string &t, string &p) {
  int l = 0, r = 0;
  vector<int> pi = precalcular_pi(p);
  bool result = false;
  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-1];
    }
  }
  return r-l == p.size();
}
```

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