Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2019

Departamento de Computación - FCEyN - UBA

String Matching

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 *text* (texto) y un string *pattern* (patrón), queremos saber si *pattern* se encuentra dentro de *text*.
- **Notación:** La función $subseq(t, d, h)$ denota al substring de t entre d y $h-1$ (inclusive). Lo abreviamos como $subseq(t, d, h)$
- proc $contiene$(in $text, pattern : seq\langle Char\rangle$, out $result$: Bool){

  Pre $\{True\}$
  Post $\{result = true \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i \leq |text| - |pattern|$

  $\wedge_L\ subseq(text, i, i + |pattern|) = pattern)\}$
  }
- ¿Cómo resolvemos este problema?

Función Auxiliar `matches`

- Implementemos una función auxiliar con la siguiente que retorna *true* si el patrón aparece a partir de una posición i del texto
- ¿Cómo lo especificamos?
- proc $matches$(in $text : seq\langle Char\rangle$, in $i : \mathbb{Z}$,
  in $pattern : seq\langle Char\rangle$, out $result$: Bool){

  Pre $\{0 \leq i < |text| - |pattern| \wedge |pattern| \leq |text|\}$
  Post $\{result = true$
  $\leftrightarrow$
  $subseq(text, i, i + |pattern|) = pattern\}$
  }

Función Auxiliar `matches()`

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

Este programa se interrumpe tan pronto como detecta una desigualdad.

Función Auxiliar `matches()`

¿Podemos escribir el mismo programa usando solo `for`?

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

¿Cuál es el tiempo de ejecución de peor caso de `matches`?

Función Auxiliar `matches()`

¿Cuál es el tiempo de ejecución de peor caso de `matches`?

```
bool matches(string &text, int i, string &pattern) {
  int k = 0;
  for (k = 0; k < pattern.size()
    && text[i+k]==pattern[k]; k++) { // O(1)
    //skip
  }
  return k == pattern.size(); // O(1)
}
```

En peor caso, el `for` se ejecuta $|pattern| + 1$ veces. Por lo tanto $T_{matches}(p) \in O(|p|)$

Búsqueda de un patrón en un texto

- **Algoritmo sencillo:** Recorrer todas las posiciones *i* de *text*, y para cada una verificar si $subseq(text, i, i + |pattern|) = pattern$.
- Podemos usar la la función auxiliar `matches` definida anteriormente.
- ¿Cómo la implementamos?

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

- ¿Cuál es el tiempo de ejecución de peor caso de `contiene`?

Búsqueda de un patrón en un texto

- ¿Cuál es el tiempo de ejecución de peor caso de `contiene`?

```
bool contiene(string &text, string &pattern) {
  int i =0;
  for (i = 0; i + pattern.size() ≤ text.size()
    && !matches(text,i,pattern); i++) { // O(|pattern|)
      // skip
  }
  return i + pattern.size() ≤ text.size(); // O(1)
}
```

- El `for` se ejecuta $|text|$-veces en peor caso.
- Por lo tanto, $T_{contiene}(text, pattern) \in O(|pattern|) * O(|text|) = O(|pattern| * |text|)$

Algoritmo de Knuth, Morris y Pratt

- En 1977, Donald Knuth, James Morris y Vaughan Pratt propusieron un algoritmo con un tiempo de ejecución de peor caso $\notin O(|text| * |pattern|)$
- **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 $subseq(t, 0, r)$.

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 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 de Knuth, Morris y Pratt Version 1.0

```
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();
}
```


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 $subseq(t, l, r+1) \neq subseq(p, 0, r-l+1)$.
- Ejemplo:

		l ↓				r ↓	$r+1$ ↓					
...	...	a	b	a	b	a	b	a	c	a	b	b
		$=$	$=$	$=$	$=$	$=$	$\neq$					
		a	b	a	b	a	c	a				
						↑ $r-l$	↑ $r-l+1$					

- ¿Hasta donde podría 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	$\langle\rangle$
abababa	$\langle\rangle$,a

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

KMP: Función π

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

KMP: Función π

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

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

Algoritmo de Knuth, Morris y Pratt

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

				l		l'			r					
				$\downarrow$		$\downarrow$			$\downarrow$					
...	...	...	...	a	b	a	b	a	b	a	c	a	b	b
						=	=	=	?					
				a	b	a	b	a	c	a				
							$\uparrow$							
							$r - l'$							

- 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 $l' = r - \pi(r - l)$.

Algoritmo de Knuth, Morris y Pratt

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

				l		l'			r					
				$\downarrow$		$\downarrow$			$\downarrow$					
...	...	...	...	a	b	a	b	a	b	a	c	a	b	b
						=	=	=	?					
						a	b	a	b	a	c	a		
									$\uparrow$					
									$r - l'$					

- 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 $l' = r - \pi(r - l)$.

Algoritmo de Knuth, Morris y Pratt

- Podemos calcular $\pi(i)$ usando una función auxiliar.
- Pero dado que usaremos muchas veces esos valores podemos precalcularlos y guardarlos en una secuencia (¿por qué?).
- Supongamos que ya tenemos una función auxiliar `calcular_pi(p)` que retorna una secuencia de enteros con los valores de π,
- Asumamos que su tiempo de ejecución de peor caso $\in O(|p|)$
- Implementemos ahora `contieneKMP`

Algoritmo de Knuth, Morris y Pratt (versión 1.0)

```
bool contieneKMP(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();
}
```

¿Qué tiempo de ejecución de peor caso tiene `contieneKMP`?

Algoritmo de Knuth, Morris y Pratt (versión 1.0)

```
bool contieneKMP(string &t, string &p) {
  int l = 0, r = 0; // O(1)
  while( r < t.size() && r-l < p.size()) { // O(1)
    if( t[r] == p[r-l] ){ // O(1)
      r++;  // O(1)
    } else if( l == r ) { // O(1)
      r++;  // O(1)
      l++;  // O(1)
    } else {
      vector<int> pi = calcular_pi(p); // O(|p|)
      l = r - pi[r-l];  // O(1)
    }
  }
  return r-l == p.size();  // O(1)
}
```

En peor caso el cuerpo del `while` se ejecutará $(|t| - |p|)$-veces. Por lo tanto, $T_{contieneKMP}(t, p) \in O((|t| - |p|) * |p|) = O(|t| * |p|)$

Algoritmo de Knuth, Morris y Pratt (versión 2.0)

¿Cómo podemos mejorar el tiempo de ejecución de peor caso?
Podemos computar la secuencia π una única vez

```
bool contieneKMP(string &t, string &p) {
  vector<int> pi = calcular_pi(p); // O(|p|)
  int l = 0, r = 0; // O(1)
  while( r < t.size() && r-l < p.size()) { // O(1)
    if( t[r] == p[r-l] ){ // O(1)
      r++;  // O(1)
    } else if( l == r ) { // O(1)
      r++;  // O(1)
      l++;  // O(1)
    } else {
      l = r - pi[r-l];  // O(1)
    }
  }
  return r-l == p.size();  // O(1)
}
```

Ahora, $T_{contieneKMP}(t, p) \in O(|t| + |p|)$

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.

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. $0 \leq i < j \leq |p|$
  2. $pi[k] = \pi(k+1)$ para $k = 0, \ldots, j-1$ (vector empieza en 0)
  3. i es la longitud del máximo bifijo para $subseq(p, 0, j)$.
  4. $0 \leq \pi(j) \leq i + 1$

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 más 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 $\langle a, b, a, b, a, c, a\rangle$

Algoritmo de Knuth, Morris y Pratt

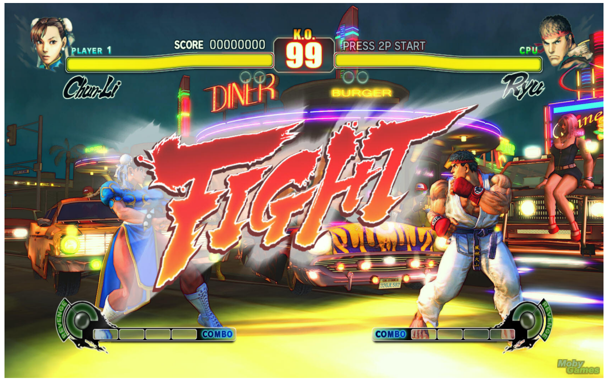
- ¡Es importante observar que sin el invariante, es muy difícil entender este algoritmo!
- ¿Cómo es una función variante adecuada para el ciclo?
  1. ¿Para el loop interno? $fv = i$
  2. ¿y para el externo? $fv = |p| - j$.
- ¿Y el tiempo de ejecución de peor caso?
  1. siempre se incrementa j
  2. i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

Recap: Algoritmo 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-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 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)

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