Algoritmos de búsqueda sobre secuencias

Algoritmos y Estructuras de Datos I

Búsqueda lineal

- Recordemos el problema de búsqueda por valor de un elemento en una secuencia.
- proc *contiene*(in $s : seq\langle\mathbb{Z}\rangle$, in $x : \mathbb{Z}$, out *result* : Bool){
  Pre $\{True\}$
  Post $\{result = true \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |s| \wedge_L s[i] = x)\}$
  }
- ¿Cómo podemos buscar un elemento en una secuencia?

Búsqueda lineal

s[0]	s[1]	s[2]	s[3]	s[4]	...	s[$\|s\|-1$]
$= x? \neq x$	$= x? \neq x$	$= x? \neq x$	$= x? \neq x$			$= x? \neq x$
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$			$\uparrow$
i	i	i	i			i

- ¿Qué invariante de ciclo podemos proponer?

  $I \equiv 0 \leq i \leq |s| \wedge_L$
  $(\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$

- ¿Qué función variante podemos usar?

  $fv = |s| - i$

Búsqueda lineal

- Invariante de ciclo:

$$I \equiv 0 \leq i \leq |s| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$$

- Función variante:

$$fv = |s| - i$$

- ¿Cómo lo podemos implementar en C++?

```
1 bool contiene(vector<int> &s, int x) {
2   int i = 0;
3   while( i < s.size() && s[i] != x ) {
4     i=i+1;
5   }
6   return i < s.size();
7 }
```

- ¿Es la implementación correcto con respecto a la especificación?

Recap: Teorema de corrección de un ciclo

- **Teorema.** Sean un predicado I y una función $fv : \mathbb{V} \to \mathbb{Z}$ (donde $\mathbb{V}$ es el producto cartesiano de los dominios de las variables del programa), y supongamos que $I \Rightarrow \text{def}(B)$. Si

  1. $P_C \Rightarrow I$,
  2. $\{I \wedge B\}\ S\ \{I\}$,
  3. $I \wedge \neg B \Rightarrow Q_C$,
  4. $\{I \wedge B \wedge v_0 = fv\}$ **S** $\{fv < v_0\}$,
  5. $I \wedge fv \leq 0 \Rightarrow \neg B$,

  ... entonces la siguiente tripla de Hoare es válida:

$$\{P_C\}\ \textbf{while B do S endwhile}\ \{Q_C\}$$

Búsqueda lineal

- Para este programa, tenemos:
  - $P_C \equiv i = 0$,
  - $Q_C \equiv (i < |s|) \leftrightarrow (\exists j : \mathbb{Z})(0 \leq j < |s| \wedge_L s[j] = x)$.
  - $B \equiv i < |s| \wedge_L s[i] \neq x$
  - $I \equiv 0 \leq i \leq |s| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$
  - $fv = |s| - i$
- Ahora tenemos que probar que:
  1. $P_C \Rightarrow I$,
  2. $\{I \wedge B\}\ S\ \{I\}$,
  3. $I \wedge \neg B \Rightarrow Q_C$,
  4. $\{I \wedge B \wedge v_0 = fv\}$ **S** $\{fv < v_0\}$,
  5. $I \wedge fv \leq 0 \Rightarrow \neg B$,

Recap: Teorema de corrección de un ciclo

1. $P_C \Rightarrow I$,
2. $\{I \wedge B\}\ S\ \{I\}$,
3. $I \wedge \neg B \Rightarrow Q_C$,
4. $\{I \wedge B \wedge v_0 = fv\}$ **S** $\{fv < v_0\}$,
5. $I \wedge fv \leq 0 \Rightarrow \neg B$,

En otras palabras, hay que mostrar que:

- I es un invariante del ciclo (punto 1. y 2.)
- Se cumple la postcondición del ciclo a la salida del ciclo (punto 3.)
- La función variante es estrictamente decreciente (punto 4.)
- Si la función variante alcanza la cota inferior la guarda se deja de cumplir (punto 5.)

Corrección de búsqueda lineal

¿I es un invariante del ciclo?

$$I \equiv 0 \leq i \leq |s| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$$

- La variable i toma el primer valor 0 y se incrementa por cada iteración hasta llegar a $|s|$.
- $\Rightarrow 0 \leq i \leq |s|$
- En cada iteración, todos los elementos a izquierda de i son distintos de x
- $\Rightarrow (\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$

Corrección de búsqueda lineal

¿Se cumple la postcondición del ciclo a la salida del ciclo?

$$I \equiv 0 \leq i \leq |s| \wedge_L (\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$$

$$Q_C \equiv (i < |s|) \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |s| \wedge_L s[i] = x)$$

- Al salir del ciclo, no se cumple la guarda. Entonces no se cumple $i < |s|$ o no se cumple $s[i] \neq x$
  - Si no se cumple $i < |s|$, no existe ninguna posición que contenga x
  - Si no se cumple $s[i] \neq x$, existe al menos una posición que contiene a x

Corrección de búsqueda lineal

¿Es la función variante estrictamente decreciente?

$$fv = |s| - i$$

- En cada iteración, se incremente en 1 el valor de i
- Por lo tanto, en cada iteración se reduce en 1 la función variante.

Corrección de búsqueda lineal

¿Si la función variante alcanza la cota inferior la guarda se deja de cumplir?

$$fv = |s| - i$$

$$B \equiv i < |s| \wedge_L s[i] \neq x$$

- Si $fv = |s| - i \leq 0$,entonces $i \geq |s|$
- Como siempre pasa que $i \leq |s|$, entonces es cierto que $i = |s|$
- Por lo tanto $i < |s|$ es falso.

Corrección de búsqueda lineal

- Finalmente,ahora que probamos que:
  1. $P_C \Rightarrow I$,
  2. $\{I \wedge B\}\ S\ \{I\}$,
  3. $I \wedge \neg B \Rightarrow Q_C$,
  4. $\{I \wedge B \wedge v_0 = fv\}\ \mathbf{S}\ \{fv < v_0\}$,
  5. $I \wedge fv \leq 0 \Rightarrow \neg B$,
- ...podemos por el teorema concluir que el ciclo termina y es correcto.

Búsqueda lineal

- Implementación:

```
1  bool contiene(vector<int> &s, int x) {
2    int i = 0;
3    while( i < s.size() && s[i] != x ) {
4      i=i+1;
5    }
6    return i < s.size();
7  }
```

- Analicemos cuántas veces itera este programa:

s	x	# iteraciones
$\langle\rangle$	1	0
$\langle 1\rangle$	1	0
$\langle 1,2\rangle$	2	1
$\langle 1,2,3\rangle$	4	3
$\langle 1,2,3,4\rangle$	4	3
$\langle 1,2,3,4,5\rangle$	-1	4

Búsqueda lineal

- ¿Cuántas veces se ejecuta el ciclo? Esto depende de
  - El tamaño de la secuencia
  - Si el valor buscado está o no contenido en la secuencia
- ¿Qué tiene que pasar para que la cantidad de ejecuciones sea máxima?
  - El elemento no debe estar contenido.
- Esto representa el **peor caso** en cantidad de iteraciones, ya que tarda mas
- Dado una secuencia cualquiera, ¿cuál es la cantidad máxima de iteraciones (el peor caso) que puede ejecutar el algortimo? En peor caso se ejecuta $|s|$ veces.

Búsqueda sobre secuencias ordenadas

- Supongamos que la secuencia está ordenada.
- proc *contieneOrdenada*(in $s : seq\langle\mathbb{Z}\rangle$, in $x : \mathbb{Z}$, out *result* : Bool){

  Pre {ordenado(s)}

  Post $\{result = True \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |s| \wedge_L s[i] = x)\}$

  }
- ¿Podemos aprovechar que la secuencia está ordenada para crear un programa más eficiente ?

Búsqueda sobre secuencias ordenadas

Podemos interrumpir la búsqueda tan pronto como verificamos que $s[i] \geq x$.

```
1  bool contieneOrdenada(vector<int> &s, int x) {
2    int i = 0;
3    while( i < s.size() && s[i] < x ) {
4      i=i+1;
5    }
6    return (i < s.size() && s[i] == x);
7  }
```

¿Cuántas veces se ejecuta el ciclo en peor caso?

Búsqueda sobre secuencias ordenadas

¿Cuántas veces se ejecuta el ciclo en peor caso?

s	x	# iteraciones (contiene)	# iteraciones (contieneOrdenada)
$\langle\rangle$	1	0	0
$\langle 1\rangle$	10	1	1
$\langle 1,2\rangle$	10	2	2
$\langle 1,2,3\rangle$	10	3	3
$\langle 1,2,3,4\rangle$	10	4	4
$\langle 1,2,3,4,5\rangle$	10	5	5
...	...	...	...
s	$x \notin s$	$\lvert s\rvert$	$\lvert s\rvert$

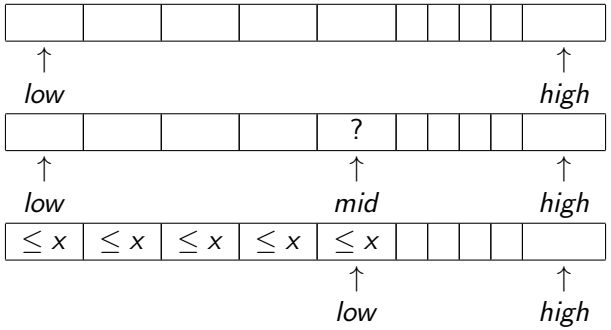
En **peor caso** (cuando el elemento no está) ambos se ejecutan la misma cantidad de veces.

Búsqueda sobre secuencias ordenadas

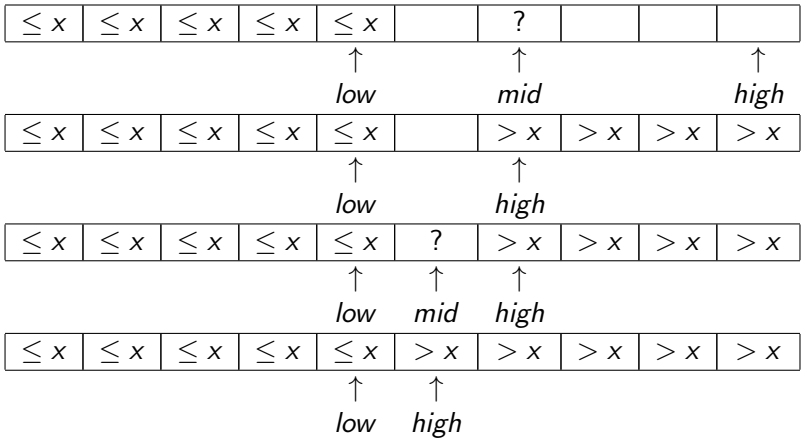
- ¿Cómo podemos aprovechar que la secuencia está ordenada para mejorar el peor caso de ejecución?
  - ¿Necesitamos iterar si $|s| = 0$? Trivialmente, $x \notin s$
  - ¿Necesitamos iterar si $|s| = 1$?Trivialmente, $s[0] == x \leftrightarrow x \in s$
  - ¿Necesitamos iterar si $x < s[0]$? Trivialmente, $x \notin s$
  - ¿Necesitamos iterar si $x > s[|s| - 1]$? Trivialmente, $x \notin s$

Búsqueda sobre secuencias ordenadas

Asumamos por un momento que $|s| > 1 \wedge_L (s[0] \leq x \leq s[|s| - 1])$

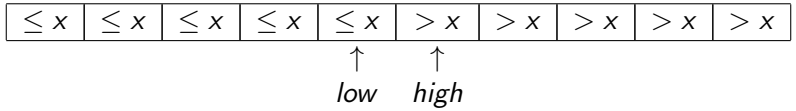


Búsqueda sobre secuencias ordenadas



Si $x \in s$, tiene que estar en la posición *low* de la secuencia.

Búsqueda sobre secuencias ordenadas



- ¿Qué invariante de ciclo podemos escribir?

$$I \equiv 0 \leq low < high < |s| \wedge_L s[low] \leq x < s[high]$$

- ¿Qué función variante podemos definir?

$$fv = high - low - 1$$

Búsqueda sobre secuencias ordenadas

```
1  bool contieneOrdenada(vector<int> &s, int x) {
2    // casos triviales
3    if (s.size()==0 ) {
4      return false;
5    } else if (s.size()==1) {
6      return s[0]==x;
7    } else if (x<s[0]) {
8      return false;
9    } else if (x>=s[s.size()-1]) {
10     return s[s.size()-1]==x;
11   } else {
12     // casos no triviales
13     ...
14   }
15 }
```

Búsqueda sobre secuencias ordenadas

```
1    } else {
2      // casos no triviales
3      int low = 0;
4      int high = s.size() - 1;
5      while( low+1 < high ) {
6        int mid = (low+high) / 2;
7        if( s[mid] <= x ) {
8          low = mid;
9        } else {
10         high = mid;
11       }
12     }
13     return s[low] == x;
14   }
15 }
```

A este algoritmo se lo denomina búsqueda binaria

Búsqueda binaria

- Veamos ahora que este algoritmo es correcto.

$$\begin{aligned} P_C &\equiv ordenada(s) \wedge (|s| > 1 \wedge_L s[0] \leq x \leq [|s| - 1]) \\ &\quad \wedge\ low = 0 \wedge high = |s| - 1 \\ Q_C &\equiv (s[low] = x) \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |s| \wedge_L s[i] = x) \\ B &\equiv low + 1 < high \\ I &\equiv 0 \leq low < high < |s| \ \wedge_L \ s[low] \leq x < s[high] \\ fv &= high - low - 1 \end{aligned}$$

Corrección de la búsqueda binaria

- ¿Es I un invariante para el ciclo?
  - El valor de *low* es siempre menor estricto que *high*
  - *low* arranca en 0 y sólo se aumenta
  - *high* arranca en $|s| - 1$ y siempre se disminuye
  - Siempre se respecta que $s[low] \leq x$ y que $x < s[high]$
- ¿A la salida del ciclo se cumple la postcondicion Q_C?
  - Al salir, se cumple que $low + 1 = high$
  - Sabemos que $s[high] > x$ y $s[low] <= x$
  - Como s está ordenada, si $x \in s$, entonces $s[low] = x$

Corrección de la búsqueda binaria

- ¿Es la función variante estrictamente decreciente?
  - Nunca ocurre que $low = high$
  - Por lo tanto, siempre ocurre que $low < mid < high$
  - De este modo, en cada iteración, o bien high es estrictamente menor, o bien low es estrictamente mayor.
  - Por lo tanto, la expresión $high - low - 1$ siempre es estrictamente menor.
- ¿Si la función variante alcanza la cota inferior la guarda se deja de cumplir?
  - Si $high - low - 1 \leq 0$, entonces $high \leq low + 1$.
  - Por lo tanto, no se cumple ($high > low + 1$), que es la guarda del ciclo

Búsqueda binaria

- ¿Podemos interrumpir el ciclo si encontramos x antes de finalizar las iteraciones?
- Una posibilidad **no recomendada** (no lo hagan en casa!):

```
1   ...
2   while( low+1 < high) {
3     int mid = (low+high) / 2;
4     if( s[mid] < x ) {
5       low = mid;
6     } else if( s[mid] > x ) {
7       high = low;
8     } else {
9       return true; // Argh!
10    }
11  }
12  return s[low] == x;
13 }
```

Búsqueda binaria

- Una posibilidad **aún peor** (ni lo intenten!):

```
1   bool salir = false;
2   while( low+1 < high && !salir ) {
3     int mid = (low+high) / 2;
4     if( s[mid] < x ) {
5       low = mid;
6     } else if( s[mid] > x ) {
7       high = mid;
8     } else {
9       salir = true; // Puaj!
10    }
11  }
12
13  return s[low] == x || s[(low+high)/2] == x;
14 }
```

Búsqueda binaria

- Si queremos salir del ciclo, el lugar para decirlo es ... la guarda!

```
1   while( low+1 < high && s[low] != x ) {
2     int mid = (low+high) / 2;
3     if( s[mid] <= x ) {
4       low = mid;
5     } else {
6       high = mid;
7     }
8   }
9   return s[low] == x;
10  }
```

- Usamos fuertemente la condición $s[low] \leq x < s[high]$ del invariante.

Búsqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	$high - low$
0	$\lvert s\rvert - 1$
1	$\cong (\lvert s\rvert - 1)/2$
2	$\cong (\lvert s\rvert - 1)/4$
3	$\cong (\lvert s\rvert - 1)/8$
$\vdots$	$\vdots$
t	$\cong (\lvert s\rvert - 1)/2^t$

- Sea t la cantidad de iteraciones necesarias para llegar a $high - low = 1$.

  $1 \cong (|s|-1)/2^t$ entonces $2^t \cong |s|-1$ entonces $t \cong \log_2(|s|-1)$.

Búsqueda binaria

- ¿Es mejor un algoritmo que ejecuta una cantidad logarítmica de iteraciones?

$\lvert s\rvert$	Búsqueda Lineal	Búsqueda Binaria
10	10	4
10^2	100	7
10^6	$1,000,000$	21
$2{,}3 \times 10^7$	$23,000,000$	25
7×10^9	$7,000,000,000$	33 (!)

- Sí! Búsqueda binaria es más eficiente que búsqueda lineal
- Pero, requiere que la secuencia esté ya ordenada.

Nearly all binary searches are broken!

http://goo.gl/Ww0Cx6

Nearly all binary searches are broken!

- En 2006 comenzaron a reportarse accesos fuera de rango a vectores dentro de la función `binarySearch` implementada en las bibliotecas estándar de Java.
- En la implementación en Java, los enteros tienen precisión finita, con rango $[-2^{31}, 2^{31} - 1]$.
- Si `low` y `high` son valores muy grandes, al calcular `k` se produce overflow.
- La falla estuvo *dormida* muchos años y se manifestó sólo cuando el tamaño de los vectores creció a la par de la capacidad de memoria de las computadoras.
- Bugfix: Computar `k` evitando el overflow:

```
int mid = low + (high-low)/2;
```

Conclusiones

- La búsqueda binaria implementada en Java estaba formalmente demostrada ...
- ... pero la demostración suponía enteros de precisión infinita (en la mayoría de los lenguajes imperativos son de precisión finita).
  - En AED1 no nos preocupan los problemas de aritmética de precisión finita (+Info: Orga1).
  - Es importante validar que las hipótesis sobre las que se realizó la demostración valgan en la implementación (aritmética finita, existencia de acceso concurrente, multi-threading, etc.)

Intervalo

Break!

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`

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

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

Función Auxiliar `iguales`

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

- `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.

```
1  bool contiene_kmp(string &t, string &p) {
2    int l = 0, r = 0;
3    bool result = false;
4    while( r < t.size() ) {
5      // Aumentar l o r
6      // Verificar si encontramos p
7    }
8    return result;
9  }
```

- ¿Cómo aumentamos l o r preservando el invariante?

Algoritmo de Knuth, Morris y Pratt

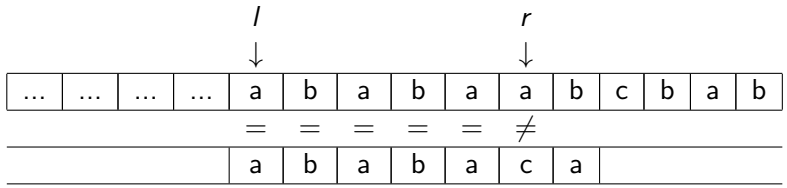
- Si $r - l = |p|$, entonces encontramos p en t.
- Si $r - l < |p|$, consideramos los siguientes casos:
  1. Si $t[r] = p[r - l]$, entonces encontramos una nueva coincidencia, y entonces incrementamos r para reflejar esta nueva situación.
  2. Si $t[r] \neq p[r - l]$ y $l = r$, entonces no tenemos un 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    bool result = false;
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 = // avanzar l
12     }
13   }
14   return r-l == p.size();
15 }
```

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:



- ¿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=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

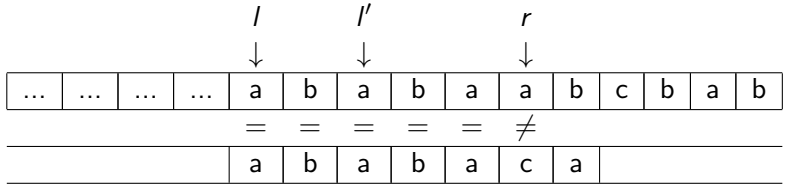
KMP: Función π

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

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	ababaca	a	1

Algoritmo de Knuth, Morris y Pratt

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



- 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

```
1  bool contiene_kmp(string &t, string &p) {
2    int l = 0, r = 0;
3    bool result = false;
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 - calcular_pi(r-l-1);
12     }
13   }
14   return r-l == p.size();
15 }
```

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

```
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    bool result = false;
5    while( r < t.size() && r-l < p.size()) {
6      if( t[r] == p[r-l] ){
7        r++;
8      } else if( l == r ) {
9        r++;
10       l++;
11     } else {
12       l = r - pi[r-l-1];
13     }
14   }
15   return r-l == p.size();
16 }
```

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.

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