# Algoritmos e Invariantes

Algoritmos y Estructuras de Datos

# Búsqueda sobre secuencias ordenadas

- La clase pasada vimos un ejemplo de especificación + algoritmos + demostración de la búsqueda lineal (`contiene()`).
- Supongamos ahora que la secuencia está ordenada.
- ¿Cómo cambiaría ahora la especificación?

```
proc contieneOrdenada(in s: seq⟨ℤ⟩, in x: ℤ): Bool){
    requiere {ordenada(s)}
    asegura {result = true ↔ (∃i : ℤ)(0 ≤ i < |s| ∧_L s[i] = x) }
```

- ¿Podemos aprovechar que la secuencia está ordenada para crear un programa más eficiente ?
- Ejercicio: Escribir el predicado ordenada(s).

# Búsqueda sobre secuencias ordenadas

- En vez de interrumpir el ciclo al encontrar el elemento, podemos interrumpirlo tan pronto como verificamos que $s[i] \geq x$.

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

- ¿Es este código realmente más eficiente que el de búsqueda lineal?
- Una forma de analizar esto es comparando "cuánto tardan" en el peor caso (i.e. cuando el elemento no está en la secuencia)

# Búsqueda sobre secuencias ordenadas

- Analicemos cómo se ejecuta el código (contando operaciones) en el peor caso.

| Función `contieneOrdenado` | $T_{exec}$ | máx.# veces |
|---|---|---|
| `int i = 0;` | $c'_1$ | 1 |
| `while( i < s.size() && s[i] < x ) {` | $c'_2$ | $1 + \|s\|$ |
| `  i=i+1;` | $c'_3$ | $\|s\|$ |
| `}` | | |
| `return (i < s.size() && s[i] == x);` | $c'_4$ | 1 |

- Sea $n$ la longitud de $s$, ¿cuál es el tiempo de ejecución en el peor caso?

$$T_{contieneOrdenado}(n) = 1 * c'_1 + (1 + n) * c'_2 + n * c'_3 + 1 * c'_4$$

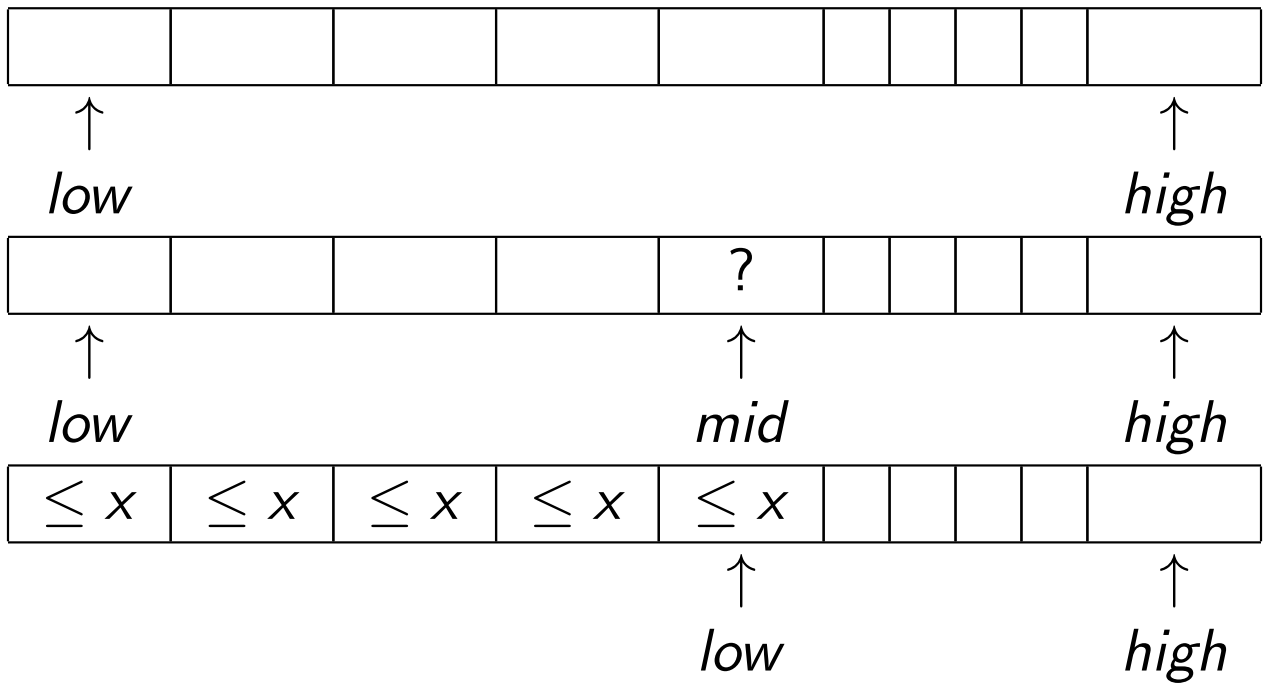
- El tiempo de ejecución de peor caso de `contiene` y **este** `contieneOrdenado` dependen linealmente de $n$.

# Búsqueda sobre secuencias ordenadas

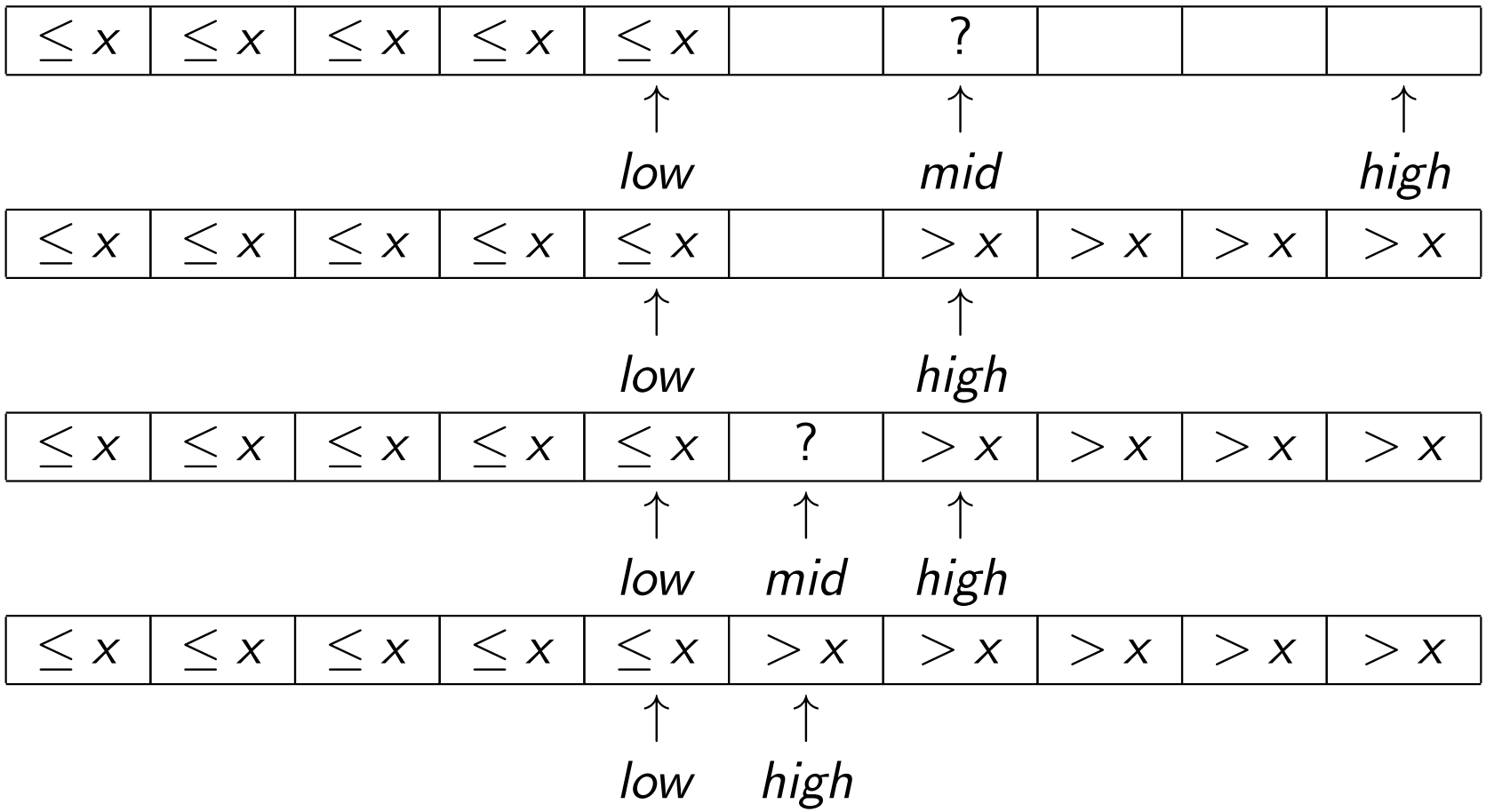
- Vamos de nuevo: ¿Podemos aprovechar el ordenamiento de la secuencia para mejorar el tiempo de ejecución de peor caso?
- Pensemos en el juego de "Adivinar un número" o "Adivinar el personaje"
  - ¿Necesitamos iterar si $|s| = 0$? Trivialmente, $x \notin s$
  - ¿Necesitamos iterar si $|s| = 1$?Trivialmente, $s[0] == x \leftrightarrow x \in s$
  - ¿Necesitamos iterar si $x < s[0]$? Trivialmente, $x \notin s$
  - ¿Necesitamos iterar si $x \geq s[|s| - 1]$? Trivialmente, $s[|s| - 1] == x \leftrightarrow x \in s$

# Búsqueda sobre secuencias ordenadas

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

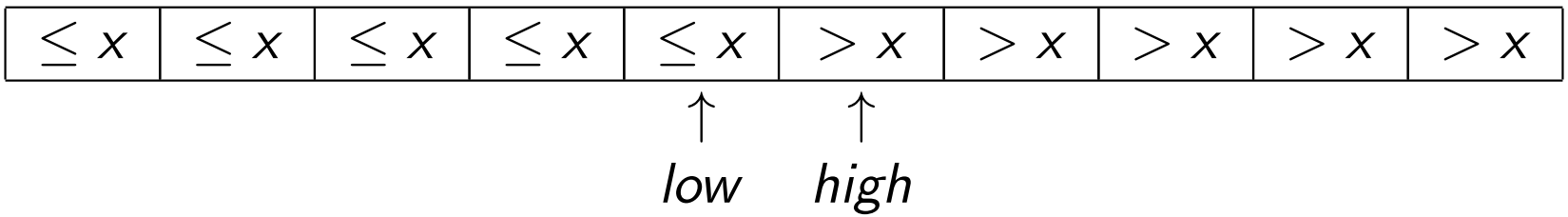


# Búsqueda sobre secuencias ordenadas




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 ordenados

```
boolean contieneOrdenada(int []s, int x) {
    // casos triviales
    if (s.length == 0 ) {
        return false;
    } else if (s.length == 1) {
        return s[0] == x;
    } else if (x<s[0]) {
        return false;
    } else if (x ≥ s[s.length-1]) {
        return s[s.length-1] == x;
    } else {
        // casos no triviales
  ○ ...
  }
}
```

# Búsqueda sobre secuencias ordenadas

```
        } else {
            // casos no triviales
            int low = 0;
            int high = s.length - 1;
            while( low+1 < high ) {
                int mid = (low+high) / 2;
                if( s[mid] ≤ x ) {
                    low = mid;
                } else {
                    high = mid;
                }
            }
            return s[low] == x;
        }
    }
```

A este algoritmo se lo denomina búsqueda binaria

# Búsqueda binaria

- Veamos ahora que este algoritmo es correcto.

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

# 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!):
- ```
  ◦ ..
  while( low+1 < high) {
    int mid = (low+high) / 2;
    if( s[mid] < x ) {
      low = mid;
    } else if( s[mid] > x ) {
      high = low;
    } else {
      return true; // Argh!
    }
  }
  return s[low] == x;
}
```
```

# Búsqueda binaria

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

- ```
    bool salir = false;
    while( low+1 < high && !salir ) {
      int mid = (low+high) / 2;
      if( s[mid] < x ) {
        low = mid;
      } else if( s[mid] > x ) {
        high = mid;
      } else {
        salir = true; // Puaj!
      }
    }

    return s[low] == x || s[(low+high)/2] == x;
  }
  ```
  ```

# Búsqueda binaria

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

- ```
  while( low+1 < high && s[low] != x ) {
    int mid = (low+high) / 2;
    if( s[mid] ≤ x ) {
      low = mid;
    } else {
      high = mid;
    }
  }
  return s[low] == x;
}
```

- 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 $high - low = 1$.

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

Luego, el tiempo de ejecución de peor caso de la búsqueda binaria es = proporcional a $\log_2 |s|$ y no proporcional a $|s|$.

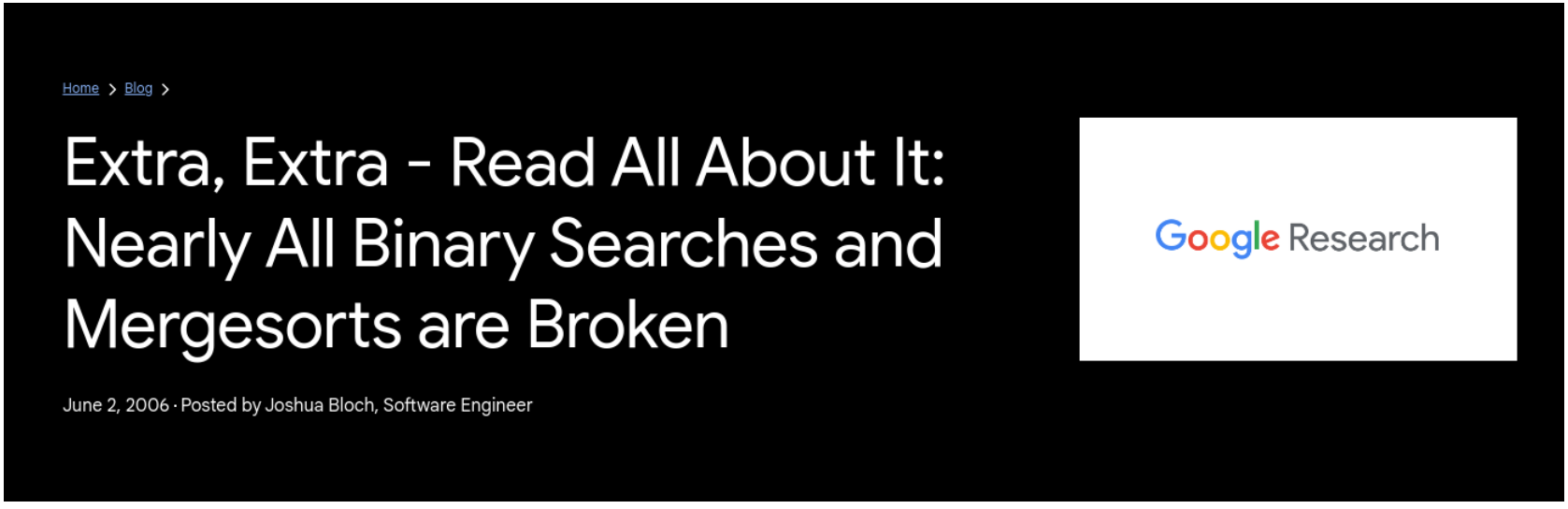
# Búsqueda binaria

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

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

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

# Bonus Track: Nearly all binary searches are broken!




`http://goo.gl/WwOCx6`

# 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 AED no nos preocupan los problemas de aritmética de precisión finita (+Info: Orga1/Sistemas Digitales).
  - 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.)

# Apareo (fusión, merge) de secuencias ordenadas

- **Problema:** Dadas dos secuencias ordenadas, fusionarlas en una única secuencia ordenada.
- El problema es importante per se y como subproblema de otros problemas importantes.
- Especificación:

$$\texttt{proc } \mathit{merge}(\texttt{in } a, b : \mathit{seq}\langle\mathbb{Z}\rangle) : \mathit{seq}\langle\mathbb{Z}\rangle \ \{$$
$$\texttt{requiere } \{\mathit{ordenada}(a) \ \wedge \ \mathit{ordenada}(b)\}$$
$$\texttt{asegura } \{\mathit{ordenada}(\mathit{result}) \ \wedge \ \mathit{mismos}(\mathit{result}, a + + b)\}$$
$$\}$$

$$\text{pred } \mathit{mismos}(s, t : \mathit{seq}\langle\mathbb{Z}\rangle)\{$$
$$(\forall x : \mathbb{Z})(\#\mathit{apariciones}(s, x) = \#\mathit{apariciones}(t, x))$$
$$\}$$

- ¿Cómo lo podemos implementar?
  - Podemos copiar los elementos de $a$ y $b$ a la secuencia $c$, y después ordenar $c$.
  - Pero no sabemos ordenar ¿Se podrá fusionar ambas secuencias en una única pasada?

# Apareo de secuencias ordenadas

Ejemplo:

- $a =$

| 1 | 3 | 5 | 7 | 9 | |
|---|---|---|---|---|---|
| ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| $i$ | $i$ | $i$ | $i$ | $i$ | $i$ |

- $b =$

| 2 | 4 | 6 | 8 | |
|---|---|---|---|---|
| ↑ | ↑ | ↑ | ↑ | ↑ |
| $j$ | $j$ | $j$ | $j$ | $j$ |

- $c =$

| ?1 | ?2 | ?3 | ?4 | ?5 | ?6 | ?7 | ?8 | ?9 | |
|---|---|---|---|---|---|---|---|---|---|
| ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| $k$ | $k$ | $k$ | $k$ | $k$ | $k$ | $k$ | $k$ | $k$ | $k$ |

# Apareo de secuencias

- ¿Qué invariante de ciclo tiene esta implementación?

$$
\begin{aligned}
I \quad \equiv \quad & \mathit{ordenada}(a) \wedge \mathit{ordenada}(b) \wedge |c| = |a| + |b| \\
\wedge \quad & ((0 \leq i \leq |a| \ \wedge \ 0 \leq j \leq |b| \ \wedge \ k = i + j) \\
\wedge_L \quad & (\mathit{mismos}(\mathit{subseq}(a, 0, i) + + \mathit{subseq}(b, 0, j), \mathit{subseq}(c, 0, k)) \\
\wedge \quad & \mathit{ordenada}(\mathit{subseq}(c, 0, k)))) \\
\wedge \quad & i < |a| \ \rightarrow_L \ (\forall t : \mathbb{Z})(0 \leq t < j \rightarrow_L b[t] \leq a[i]) \\
\wedge \quad & j < |b| \ \rightarrow_L \ (\forall t : \mathbb{Z})(0 \leq t < i \rightarrow_L a[t] \leq b[j])
\end{aligned}
$$

- ¿Qué función variante debería tener esta implementación?

$$fv = |a| + |b| - k$$

# Apareo de secuencias

```
int[] merge(int[] a, int b[]) {
        int[] c = new int[a.length+b.length];
        int i = 0; // Para recorrer a
        int j = 0; // Para recorrer b
        int k = 0; // Para recorrer c

        while( k < c.length ) {
                if( /*Si tengo que avanzar i */ ) {
                        c[k++] = a[i++];
                } else if(/* Si tengo que avanzar j */)  {
                        c[k++] = b[j++];
                }
        }
        return c;
}
```

- ¿Cuándo tengo que avanzar $i$? Cuando $j$ está fuera de rango ó cuando $i$ y $j$ están en rango y $a[i] < b[j]$
- ¿Cuándo tengo que avanzar $j$? Cuando no tengo que avanzar $i$

# Apareo de secuencias

```
int[] merge(int[] a, int b[]) {
        int[] c = new int[a.length+b.length];
        int i = 0; // Para recorrer a
        int j = 0; // Para recorrer b
        int k = 0; // Para recorrer c

        while( k < c.length ) {
                if( j>=b.length || ( i<a.length() && a[i] < b[j] ) ) {
                        c[k++] = a[i++];

                } else   {
                        c[k++] = b[j++];

                }
        }
        return c;
}
```

# Apareo de secuencias

- Al terminar el ciclo, ¿ya está la secuencia $c$ con los valores finales?
- ¿Cuál es el tiempo de ejecución de peor caso de `merge`?
- Sea $n = |c| = |a| + |b|$
- El `while` se ejecuta $n + 1$ veces.
- Por lo tanto, $T_{merge}(n) \in O(n)$

# Bibliografía


- David Gries - The Science of Programming
  - Chapter 16 - Developing Invariants (Linear Search, Binary Search)
- Cormen et al. - Introduction to Algorithms
  - Chapter 2.2 -Analyzing algorithms
  - Chapter 3 - Growth of Functions