# Repaso - TADs + Algoritmos de búsqueda sobre secuencias

Algoritmos y Estructuras de Datos







## Especifiquemos un TAD

### Grafo

```
TAD Grafo<T> {
        obs vertices: conj⟨ver_ID⟩
        obs data: dict(ver_ID, T)
        obs aristas: dict(ver_ID, conj⟨ver_ID⟩)

        proc grafoVacío(): Grafo<T>
                asegura {res.data = {} ∧ res.aristas = {} ∧ res.vertices = {}}

        proc agregarVertice(inout g: Grafo<T>, in d: T, in a: conj⟨ver_ID⟩): ver_ID
                requiere {todosIn(g.vertices, a) ∧ g = G_0}
                asegura {res ∉ G_0.vertices ∧ res ∈ g.vertices ∧_L g.data = setKey(G_0.data, res, d) ∧
                        g.aristas = setKey(G_0.aristas, res, a)}

        proc eliminarVertice(inout g: Grafo<T>, in v: ver_ID)
                requiere {v ∈ g.data ∧ g = G_0}
                asegura {g.data = delKey(G_0.data, v) ∧
                        g.vertices = G_0.vertices − {v} ∧
                        g.aristas = delKey(G_0.aristas, v)}
                asegura {noExisteComoValor(d.aristas, v) }

        proc hayCamino(in g: Grafo<T>, in v_1, v_2 : ver_ID)): T):Bool
                requiere {v_1 ∈ g.vertices ∧ v_2 ∈ g.vertices}
                asegura {res = true ↔ (∃s : seq⟨ver_ID⟩)(todosIn(g.vertices, s)∧ | s |≥ 2∧_L
                                s[0] = v_1 ∧ s[| s | −1] = v_2 ∧
                                (∀i : ℤ)(1 ≤ i <| s | ∧ →_L s[i] ∈ g.aristas[s[i − 1]])) }
```

## Búsqueda sobre secuencias ordenadas

- Hace unas clases vimos un ejemplo de espcificación + algoritmos + demostración de la búsqueda lineal (`contiene()`).
- Supongamos ahora que la secuencia está ordenada.
- ¿Cómo cambiaría ahora la especificación?

  proc contieneOrdenada(in s: $seq\langle\mathbb{Z}\rangle$, in x: $\mathbb{Z}$): Bool){
  requiere {ordenada($s$)}
  asegura $\{result = true \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |s| \wedge_L s[i] = x)\ \}$

- ¿Podemos aprovechar que la secuencia está ordenada para crear un programa más eficiente ?
- 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 cómo corren sobre el peor caso (i.e. 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 $n$.

TADs

Búsqueda sobre secuencias ordenadas

Especificación

Programa 1

Programa 2: Búsqueda binaria

Puede fallar

Apareo/Merge

Especificación

Pensemos el algoritmo

Pensemos el invariante



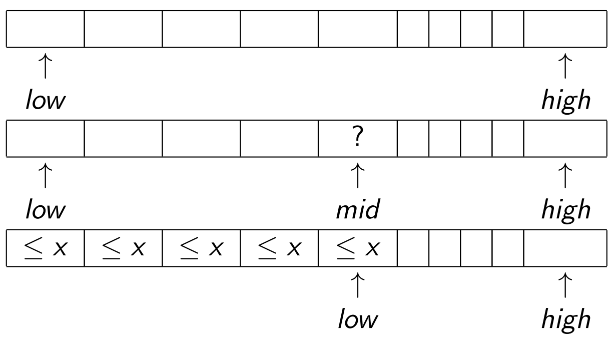
## Búsqueda sobre secuencias ordenadas

- 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, $s[|s| - 1] == x \leftrightarrow x \in s$



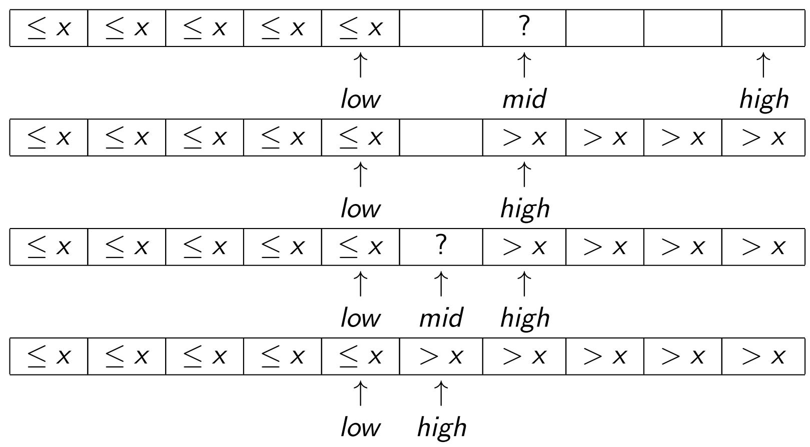
## Búsqueda sobre secuencias ordenadas

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






## Búsqueda sobre secuencias ordenadas




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

## Búsqueda sobre secuencias ordenadas

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $\leq x$ | $\leq x$ | $\leq x$ | $\leq x$ | $\leq x$ | $> x$ | $> x$ | $> x$ | $> x$ | $> x$ |
| | | | | $\uparrow$ *low* | $\uparrow$ *high* | | | | |

- ¿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.

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

## 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!):


```
 o..
 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 | $\|s\| - 1$ |
| 1 | $\cong (\|s\| - 1)/2$ |
| 2 | $\cong (\|s\| - 1)/4$ |
| 3 | $\cong (\|s\| - 1)/8$ |
| $\vdots$ | $\vdots$ |
| $t$ | $\cong (\|s\| - 1)/2^t$ |

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

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

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



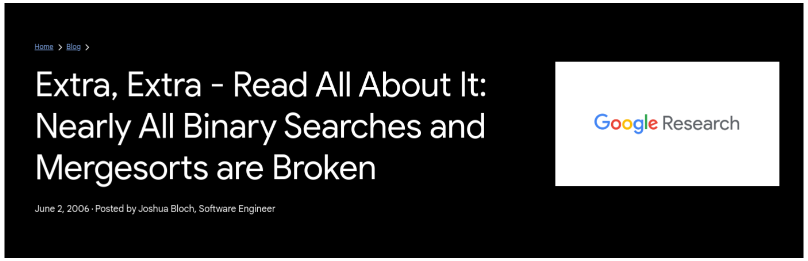
## Búsqueda binaria

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

| $\|s\|$ | 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/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 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:

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

$$
\begin{array}{l}
\text{pred}\ mismos(s, t : seq\langle\mathbb{Z}\rangle)\{ \\
\quad (\forall x : \mathbb{Z})(\#apariciones(s, x) = \#apariciones(t, x)) \\
\}
\end{array}
$$

- ¿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:

- $a = \begin{array}{|c|c|c|c|c|c} \hline 1 & 3 & 5 & 7 & 9 & \\ \hline \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} \\ \multicolumn{1}{c}{i} & \multicolumn{1}{c}{i} & \multicolumn{1}{c}{i} & \multicolumn{1}{c}{i} & \multicolumn{1}{c}{i} & \multicolumn{1}{c}{i} \end{array}$
- $b = \begin{array}{|c|c|c|c|c} \hline 2 & 4 & 6 & 8 & \\ \hline \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} \\ \multicolumn{1}{c}{j} & \multicolumn{1}{c}{j} & \multicolumn{1}{c}{j} & \multicolumn{1}{c}{j} & \multicolumn{1}{c}{j} \end{array}$
- $c = \begin{array}{|c|c|c|c|c|c|c|c|c|c} \hline ?1 & ?2 & ?3 & ?4 & ?5 & ?6 & ?7 & ?8 & ?9 & \\ \hline \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{\uparrow} \\ \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} & \multicolumn{1}{c}{k} \end{array}$

## Apareo de secuencias

- ¿Qué invariante de ciclo tiene esta implementación?

$$
\begin{array}{rl}
I \equiv & ordenada(a) \land ordenada(b) \land |c| = |a| + |b| \\
\land & ((0 \le i \le |a| \ \land \ 0 \le j \le |b| \ \land \ k = i + j) \\
\land_L & (mismos(subseq(a,0,i) + +subseq(b,0,j), subseq(c,0,k)) \\
\land & ordenada(subseq(c,0,k)))) \\
\land & i < |a| \ \rightarrow_L \ (\forall t : \mathbb{Z})(0 \le t < j \rightarrow_L b[t] \le a[i]) \\
\land & j < |b| \ \rightarrow_L \ (\forall t : \mathbb{Z})(0 \le t < i \rightarrow_L a[t] \le b[j])
\end{array}
$$

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