

Algoritmos de búsqueda sobre secuencias

Algoritmos y Estructuras de Datos

1

Búsqueda lineal

- El problema de búsqueda por valor de un elemento en una secuencia es uno de los problemas fundamentales de la Informática.
- Vamos a aprovecharlo para aplicar el Teorema del Invariante y explorar su relación con el diseño de algoritmos
- Especificado formalmente:

```
proc contiene(in s : seq(Z), in x : Z, out result : Bool){
    Pre { True }
    Post { result = true ↔ (∃i : Z)(0 ≤ i < |s| ∧ s[i] = x) }
}
```
- ¿Cómo podemos buscar un elemento en una secuencia?

2

Búsqueda lineal

s[0]	s[1]	s[2]	s[3]	s[4]	...	s[s - 1]
= x? ≠ x	= x? ≠ x	= x? ≠ x	= x? ≠ x			= x? ≠ x
↑	↑	↑	↑			↑
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?

$$fv = |s| - i$$

3

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 Java?

```
boolean contiene(int []s, int x) {
    int i = 0;
    while (i < s.length && s[i] != x) {
        i = i + 1;
    }
    return i < s.length;
}
```

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

4

Recap: Teorema de corrección de un ciclo

► **Teorema.** Sean un predicado I y una función $fv : \mathbb{V} \rightarrow \mathbb{Z}$ (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:

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

5

Búsqueda lineal

► Para este ciclo, 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. $I \wedge fv \leq 0 \Rightarrow \neg B$,

6

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.)

7

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
- $\Rightarrow (\forall j : \mathbb{Z})(0 \leq j < i \rightarrow_L s[j] \neq x)$

8

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

9

Corrección de búsqueda lineal

¿Es la función variante estrictamente decreciente?

$$fv = |s| - i$$

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

10

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.

11

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.

12

Búsqueda lineal

► Implementación:

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

- ¿Es bueno este programa?
- ¿qué quiere decir bueno?
- ¿Tarda mucho? ¿tarda demasiado?
- ¿usa mucha memoria?
- Vamos a ver esto con mucho más cuidado dentro de algunas clases
- Mientras tanto.....

13

Búsqueda lineal

► Implementación:

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

► Analicemos cuántas veces va a iterar 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	5

14

Búsqueda lineal

- ¿De qué depende la cantidad de veces que se ejecuta el ciclo?
 - Del tamaño de la secuencia
 - De si el valor buscado está o no contenido en la secuencia
- ¿Qué tiene que pasar para que el tiempo de ejecución sea el máximo posible?
 - El elemento no debe estar contenido en la secuencia.
- Esto representa el **peor caso** en tiempo de ejecución.

15

Búsqueda lineal

- Dada una secuencia cualquiera, ¿cuál es el tiempo máximo (i.e. el peor caso) que puede tardar en ejecutar el programa?

Función contiene	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();	c_4	1

- ¿Cuál es el tiempo máximo de ejecución para una secuencia s?
Sea n la longitud de s

$$T_{contiene}(n) = 1 * c_1 + (1 + n) * c_2 + n * c_3 + 1 * c_4$$

16

Búsqueda sobre secuencias ordenadas

- Supongamos ahora que la secuencia está **ordenada**.
- `proc contieneOrdenada(in s : seq<ℤ>, in x : ℤ, out result : Bool){`
`Pre {ordenada(s)}`
`Post {result = true ↔ (∃i : ℤ)(0 ≤ i < |s| ∧ s[i] = x)}`
`}`
- ¿Podemos aprovechar que la secuencia está ordenada para crear un programa más **eficiente**?
- Ejercicio: Escribir el predicado `ordenada(s)`.

17

Búsqueda sobre secuencias ordenadas

Podemos interrumpir la búsqueda 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);
}
```

¿Cuál es el tiempo de ejecución de peor caso?

18

Búsqueda sobre secuencias ordenadas

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

Función <code>contieneOrdenado</code>	T_{exec}	máx. # veces
<code>int i = 0;</code>	c'_1	1
<code>while(i < s.size() && s[i] < x) {</code>	c'_2	$1 + s $
<code> i=i+1;</code>	c'_3	$ s $
<code>}</code>		
<code>return (i < s.size() && s[i] == x);</code>	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 `contieneOrdenado` está acotado por la misma función $c * n$.

19

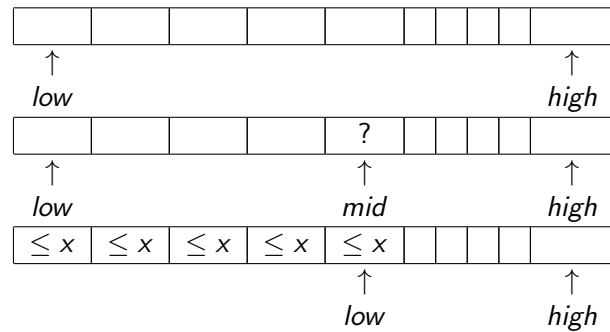
Búsqueda sobre secuencias ordenadas

- ¿Podemos aprovechar el ordenamiento de la secuencia para mejorar el tiempo de ejecución de peor caso?
- Pensemos en el juego de "Adivinar un número." "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, $s[|s| - 1] == x \leftrightarrow x \in s$**

20

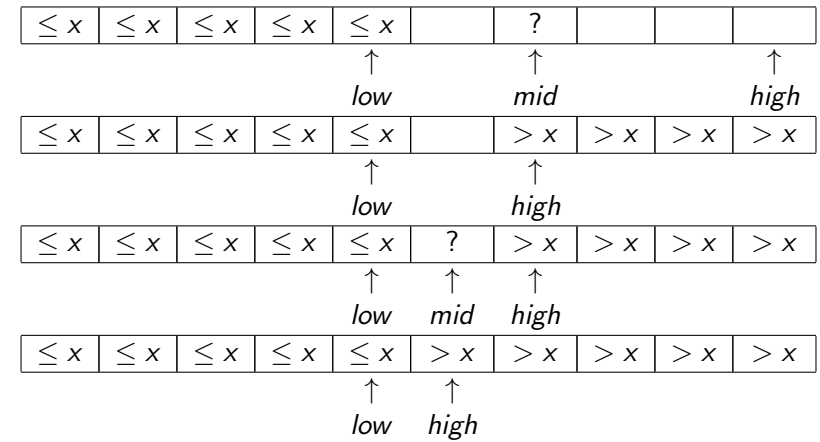
Búsqueda sobre secuencias ordenadas

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



21

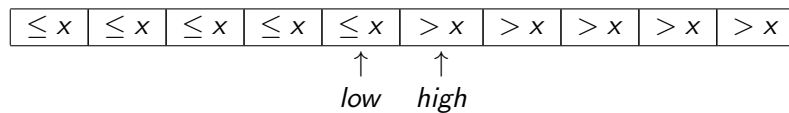
Búsqueda sobre secuencias ordenadas



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

22

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?

$$fv = high - low - 1$$

23

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
        ...
    }
}
```

24

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**

25

Búsqueda binaria

- Veamos ahora que este algoritmo es correcto.

$$P_C \equiv \text{ordenada}(s) \wedge (|s| > 1 \wedge_L s[0] \leq x \leq s[|s| - 1]) \\ \wedge \text{low} = 0 \wedge \text{high} = |s| - 1$$

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

$$B \equiv \text{low} + 1 < \text{high}$$

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

$$fv = \text{high} - \text{low} - 1$$

26

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 $s[\text{low}] \leq x$ y que $x < s[\text{high}]$
- ¿A la salida del ciclo se cumple la postcondición Q_C ?
 - Al salir, se cumple que $\text{low} + 1 = \text{high}$
 - Sabemos que $s[\text{high}] > x$ y $s[\text{low}] \leq x$
 - Como s está ordenada, si $x \in s$, entonces $s[\text{low}] = x$

27

Corrección de la búsqueda binaria

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

28

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;
}
```

29

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;
}
```

30

## 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.

31

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 $high - low = 1$.

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

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

32

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
7×10^9	7,000,000,000	33 (!)

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

33

Bonus Track: Nearly all binary searches are broken!



34

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;
```

35

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).
 - 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.)

36

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:

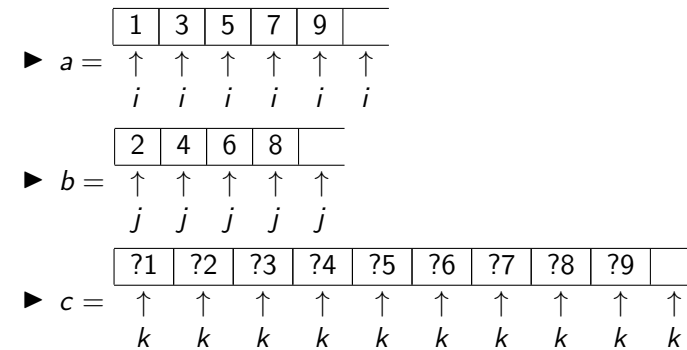

```
proc merge(in a, b : seq<Z>, out result : seq<Z>){
  Pre {ordenada(a) ∧ ordenada(b)}
  Post {ordenada(result) ∧ mismos(result, a ++ b)}
}
```

```
pred mismos(s, t : seq<Z>){
  (∀x : Z)(#apariciones(s, x) = #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**?

37

Apareo de secuencias ordenadas

Ejemplo:



38

Apareo de secuencias

- ¿Qué invariante de ciclo tiene esta implementación?

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

- ¿Qué función variante debería tener esta implementación?

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

39

Apareo de secuencias

```
vector<int> merge(vector<int> &a, vector<int> &b) {
  vector<int> c(a.size()+b.size());
  int i = 0; // Para recorrer a
  int j = 0; // Para recorrer b
  int k = 0; // Para recorrer c

  while( k < c.size() ) {
    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*

40

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, $T_{merge}(n) \in O(n)$

41

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

42