



Ayudantia 1



Inducción

Principio de Inducción Simple (PIS)

Sea P una propiedad sobre elementos de N . Si se cumple que:

- $P(n_0)$ es verdadero ($n_0 \in N$ cumple la propiedad P) **BI**
- Si $P(n)$, entonces $P(n+1)$ (cada vez que n cumple la propiedad $n+1$ también la cumple) **Paso inductivo**

Entonces todos los elementos de N a partir de n_0 cumplen la propiedad

Paso inductivo

Si $P(n) \rightarrow \text{HI}$ (Es lo que asumimos)
entonces $P(n+1) \rightarrow \text{TI}$ (Es lo que debemos demostrar)

*Importante: **NUNCA** partir desde la TI y llegar a la HI

Ejemplo Inducción:

Por demostrar: **6 divide a $n^3 - n$** para todo **n** natural.

1. BI:

2. HI:

3. TI:

Ejemplo Inducción:

Por demostrar: **6 divide a $n^3 - n$** para todo n natural.

1. BI: Para $n = 1$;

$$1^3 - 1 = 0 = 0 * 6 \qquad 6 \text{ divide a } 1^3 - 1$$

1. HI: Asumimos que la afirmación se cumple para un número N .

$$6 \text{ divide a } N^3 - N$$

1. TI: Demostramos que la afirmación se cumple para $N + 1$.

$$\begin{aligned}(N + 1)^3 - (N + 1) &= (N + 1)(N^2 + 2N) \\ &= N^3 + 3N^2 + 2N \\ &= (N^3 - N) + (3N^2 + 3N) \\ &= 6k + 3N(N + 1) \\ &= 6k + 6k'\end{aligned}$$

$$(N + 1)^3 - (N + 1) = 6k''$$

Principio de Inducción por curso de valores (PICV)

Sea P una propiedad sobre elementos de N . Si se cumple que:

- $\forall k \in N, k < n, P(k) \text{ es verdadero} \Rightarrow P(n) \text{ es verdadero}$

Entonces P es verdadero para todos los elementos de N .

En palabras simples “Suponemos que para todo numero menor que n se cumple la propiedad P , si n también la cumple entonces se cumple para todos los naturales”

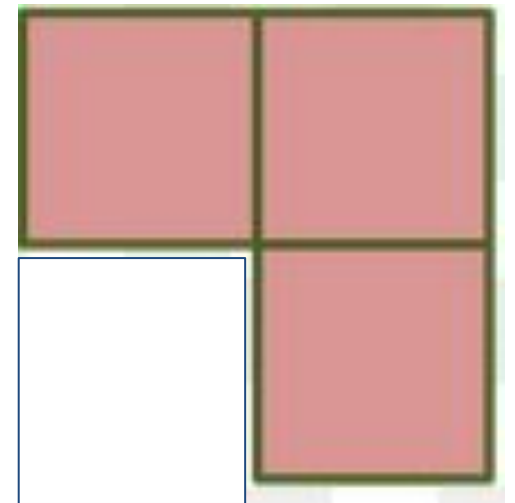
Ejemplo Inducción:

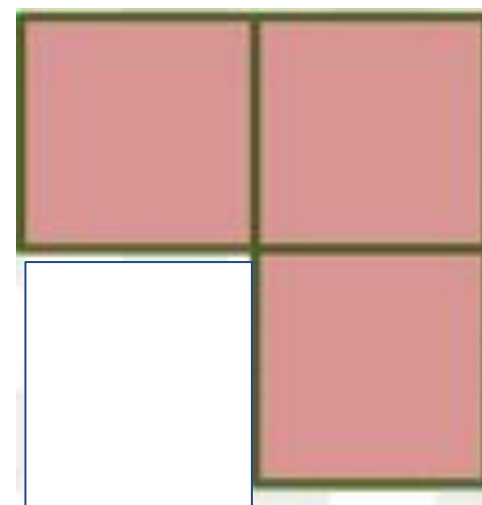
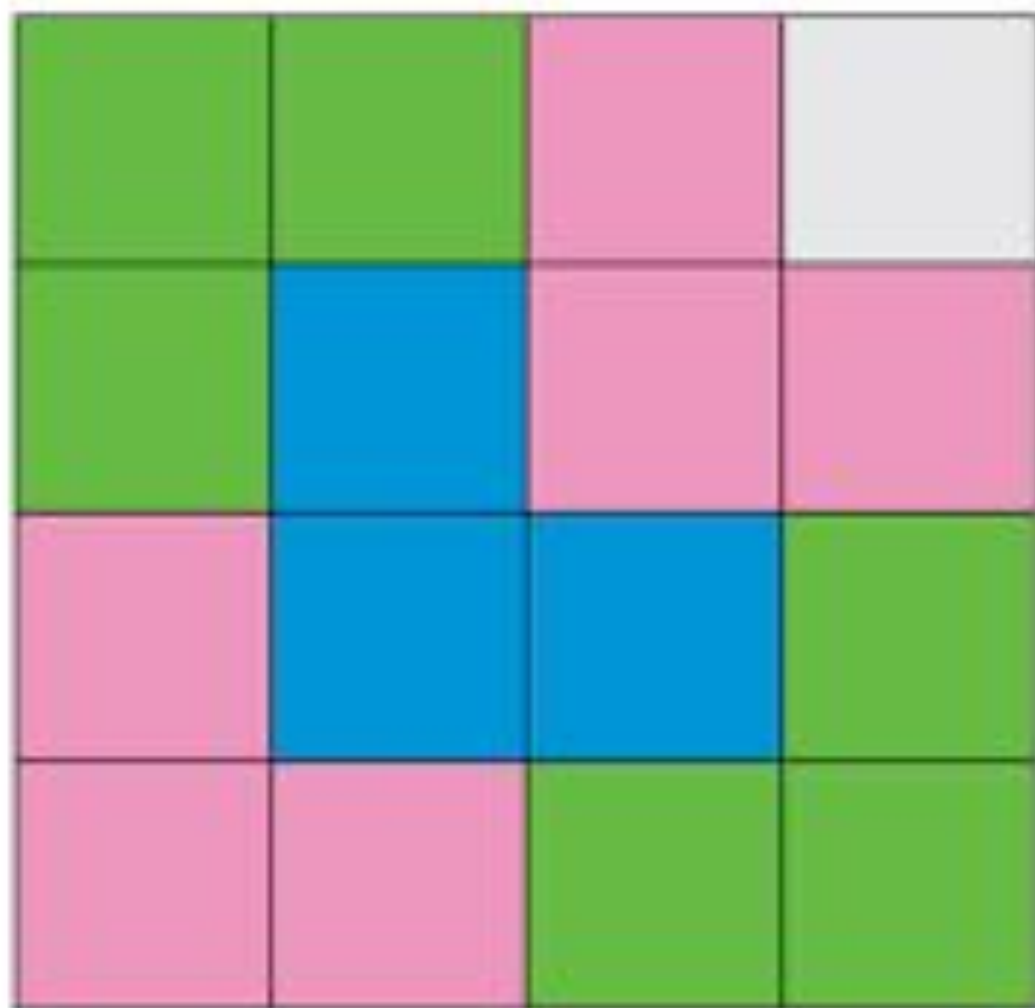
Por demostrar: Si se quita cualquier cuadrado de un tablero de ajedrez de $2^n \times 2^n$, el tablero restante puede cubrirse completamente con L-trominós. Donde un trominó es una pieza de dominó de 3 cuadrados.

1. BI:

2. HI:

3. TI:





Ejemplo Inducción:

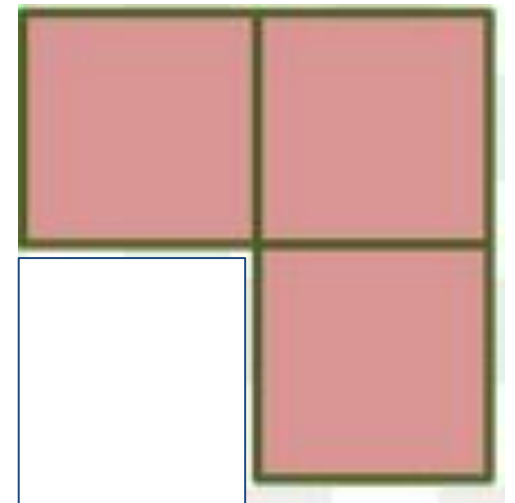
Por demostrar: Si se quita cualquier cuadrado de un tablero de ajedrez de $2^n \times 2^n$, el tablero restante puede cubrirse completamente con L-trominós. Donde un trominó es una pieza de dominó de 3 cuadrados.

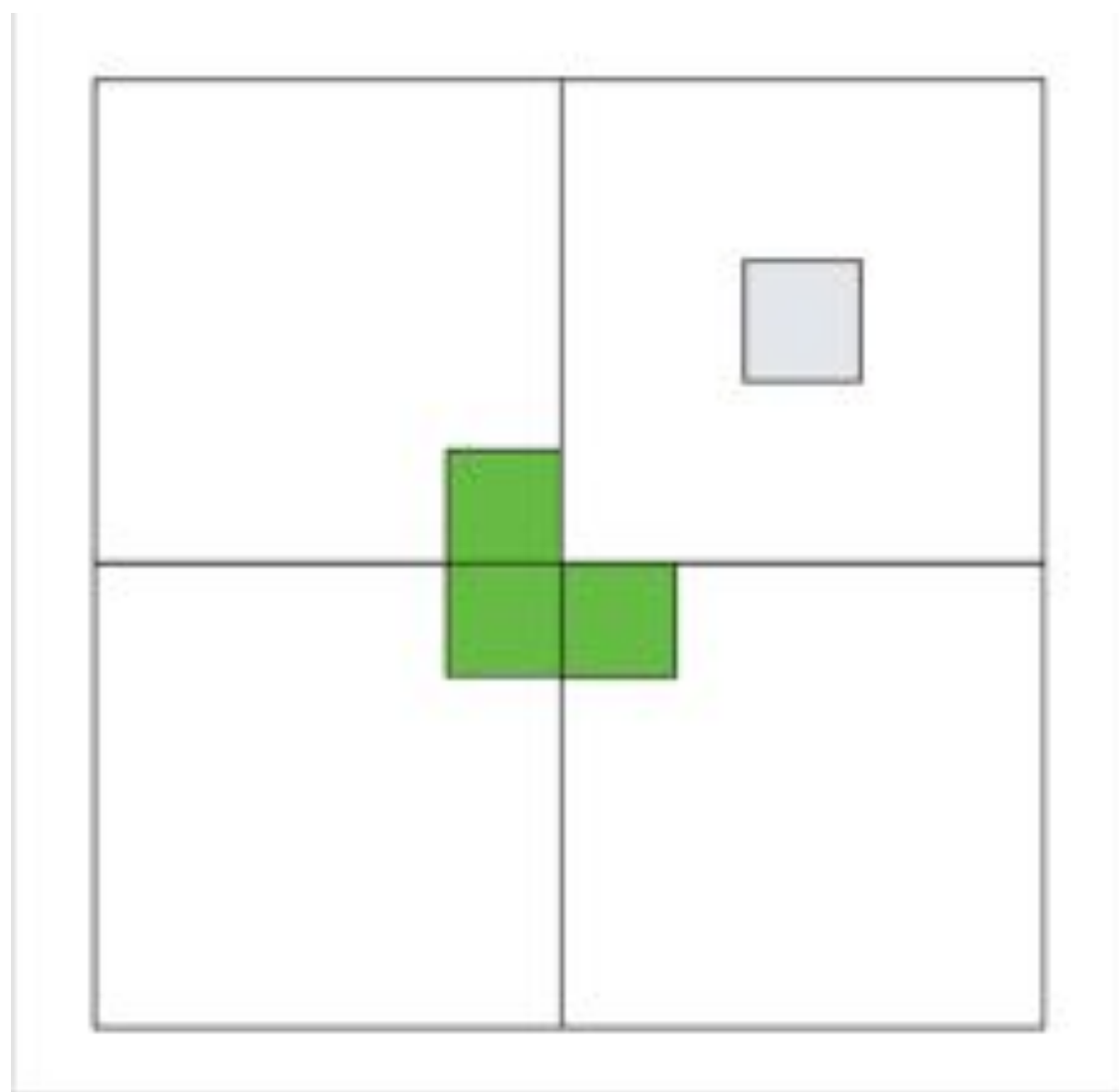
1. **BI:** Para $n = 1$.

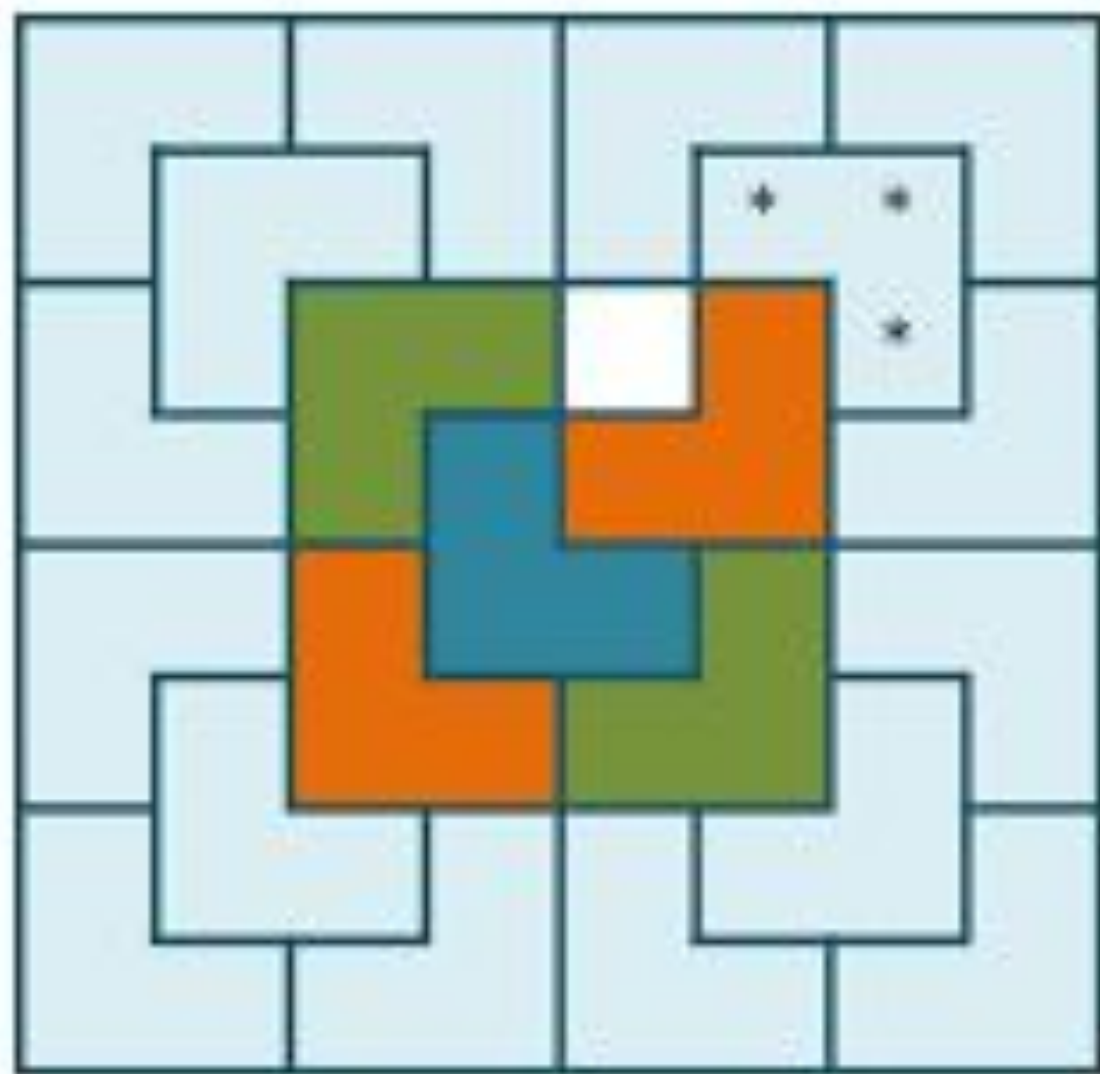
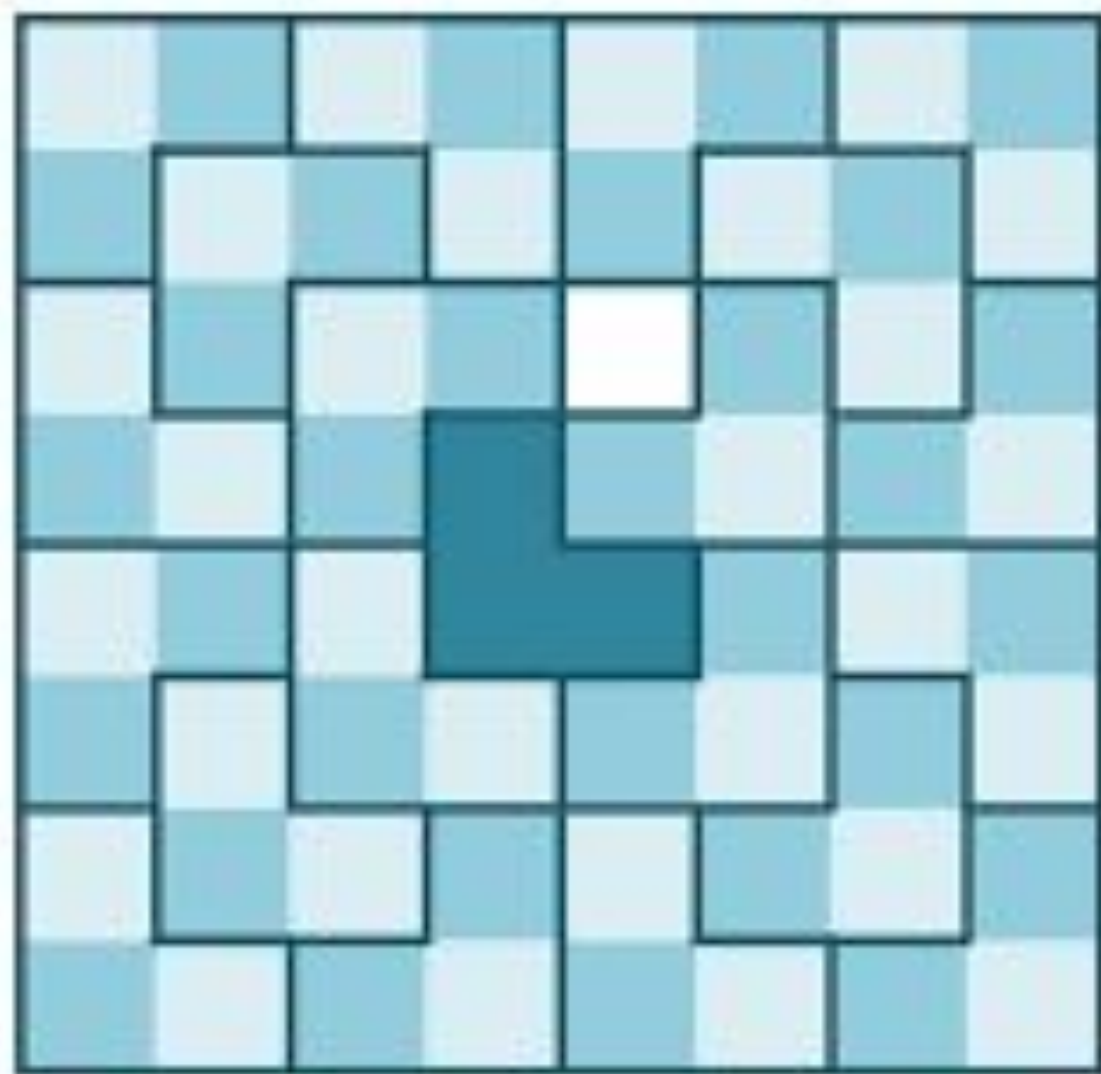
Un tablero de $2^1 \times 2^1$, al quitarle una pieza queda como un trominó.


2. **HI:** Supongamos que la propiedad se cumple para todo $k \in \mathbb{N}$, tal que $k < n$.

3. **TI:** Demostraremos que la propiedad se cumple para n









Notación Asintótica

Para qué???

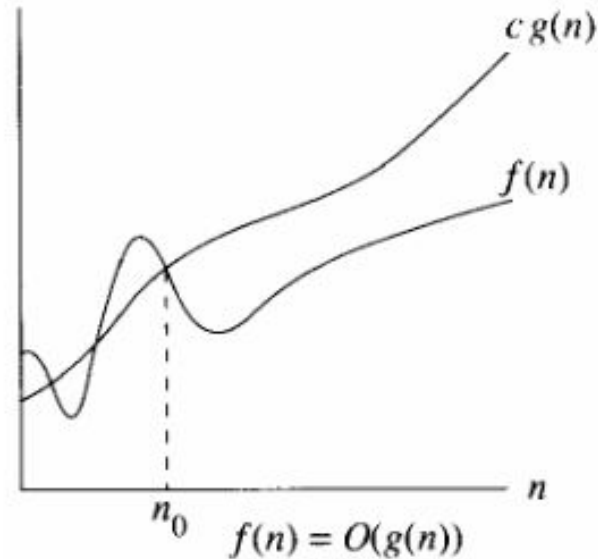
- Determinar tiempo de respuesta de nuestro algoritmo (runtime)
- Determinar recursos computacionales
- Entender cómo va a evolucionar el costo de nuestra función al hacer crecer cada vez más el tamaño del input

Nos permite elegir algoritmos más eficientes. Sin embargo, muchas veces el análisis no es trivial

Notación O

Dada una función $g(n)$, denotamos como $O(g(n))$ al conjunto de funciones tales que:

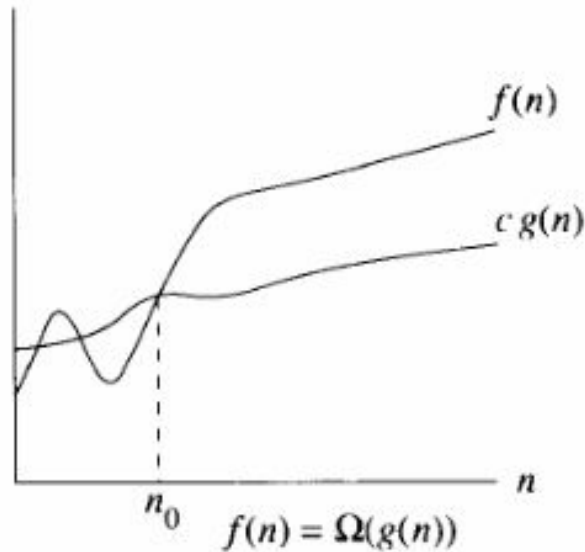
$$O(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \text{ constante positiva y } n_0 \in \mathbb{N} : f(n) \leq cg(n), \forall n \geq n_0\}$$



Notación Omega

Dada una función $g(n)$, denotamos al conjunto de funciones $\Omega(g(n))$ de la siguiente forma:

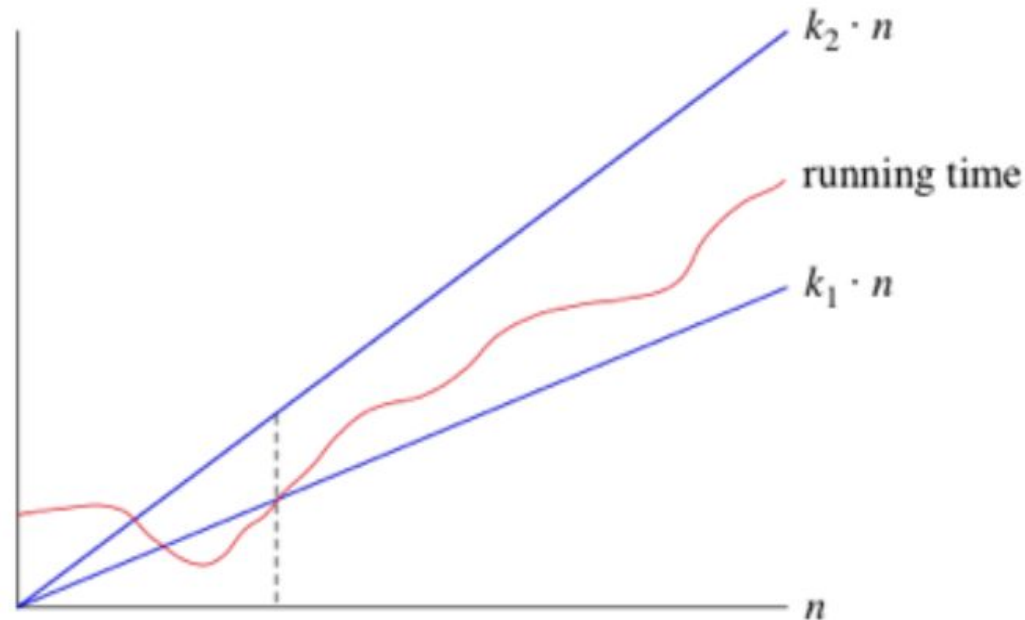
$$\Omega(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \text{ constante positiva y } n_0: 0 < cg(n) \leq f(n), \forall n \geq n_0\}$$



Notación Theta

Diremos que $f(n) \in \Theta(g(n))$ si $f(n)$ pertenece tanto a $O(g(n))$ como a $\Omega(g(n))$

$$\Theta(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1, c_2 \text{ constantes positivas, } n_0: 0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$



$T(n) = 1$	---- $> O(1)$ ----> Orden Constante
$T(n) = \log_2 n$	---- $> O(\log n)$ ----> Orden Logaritmico
$T(n) = an + b$	---- $> O(n)$ ----> Orden Lineal
$T(n) = n \log_2 n$	---- $> O(n)$ ----> Orden $n \log_2 n$
$T(n) = an^2 + bn + c$	---- $> O(n^2)$ ----> Orden Cuadratico
$T(n) = an^3 + bn^2 + c$	---- $> O(n^3)$ ----> Orden Cubico
$T(n) = n^m, m = 0, 1, 2, 3 \dots$	---- $> O(n^2)$ ----> Orden Polinomial
$T(n) = c^n, c > 1$	---- $> O(n^3)$ ----> Orden Exponencial
$T(n) = n!$	---- $> O(n^2)$ ----> Orden Factorial



Correctitud

- Un algoritmo se dice que es **correcto** si, para todo input válido, se cumple que:
 - El algoritmo termina en tiempo finito.
 - Se obtiene el resultado esperado.
- Para demostrar la correctitud de un algoritmo se suele utilizar **inducción**, debido a su buena compatibilidad con problemas de tamaño creciente.

Ejemplo Correctitud: Bubble Sort

Bubble Sort, es un algoritmo que ordena un arreglo realizando intercambios entre pares vecinos desordenados hasta que el arreglo está ordenado.

1. Se compara una posición con la siguiente, y si el número siguiente es menor, entonces se intercambian. Se repite esta acción desde la primera hasta la penúltima posición.
1. Se repite el paso 1. hasta que se realice una iteración sin ningún intercambio.

Ejemplo Correctitud: Bubble Sort

1º Iteración

2	1	8	6
---	---	---	---

1	2	8	6
---	---	---	---

1	2	8	6
---	---	---	---

1	2	6	8
---	---	---	---

Ejemplo Correctitud: Bubble Sort

2º Iteración

1	2	6	8
---	---	---	---

1	2	6	8
---	---	---	---

1	2	6	8
---	---	---	---

1	2	6	8
---	---	---	---

Ejemplo Correctitud: Bubble Sort

Inducción:

1. **BI:** Como base de inducción se elegirá a un arreglo de largo 1. Como tiene un solo elemento, está ordenado.
1. **HI:** Bubble Sort ordena todo arreglo de largo n .
1. **TI:** Bubble Sort ordena todo arreglo de largo $n + 1$.

La tesis se demuestra utilizando la hipótesis.

Ejemplo Correctitud: Bubble Sort

TI: Se demuestra que Bubble Sort es correcto para todo arreglo de largo $n + 1$, utilizando la hipótesis.

Se observa que en Bubble sort, luego de la primera iteración de intercambios, el máximo del arreglo quedará en la última posición, y se mantendrá en esa posición, ya que, al ser el mayor, nunca se cumplirá la condición de intercambio con otro número en la posición anterior.

Por lo que, siendo A un arreglo de largo $n + 1$, $\max(A)$ el máximo del arreglo A , y A' el mismo arreglo pero sacando a $\max(A)$. Se tiene que:

$$BS(A) = BS(A') + [\max(A)]$$

Luego, se observa que A' será de largo n , porque se removi6 el máximo, entonces por **HI** se deduce que $BS(A')$ entregará un arreglo ordenado. Y como el agregarle $\max(A)$ al final del arreglo mantendrá el orden correcto, se demuestra que $BS(A)$ entrega un arreglo ordenado.

Ejemplo Correctitud: Bubble Sort

- Se demostró que Bubble Sort ordena arreglos de cualquier tamaño, pero.
¿Se demostró que Bubble Sort siempre termina?
- La condición de término de Bubble Sort es que no se produzcan intercambios en una iteración, y esto ocurre si y sólo si el arreglo está ordenado.
Como para cualquier input válido se consiguen arreglos ordenados, Bubble Sort siempre termina. Y como los inputs válidos siempre serán arreglos finitos, se termina en tiempo finito.

Ejemplo Complejidad: Bubble Sort

Cual es el mejor caso?

Cual es el peor caso?

The background of the slide is a large, irregular watercolor splash. It features a gradient from a vibrant green at the top to a deep blue at the bottom, with various white and grey splatters and droplets scattered around the edges, giving it a textured, artistic appearance.

Selection sort & Insertion sort



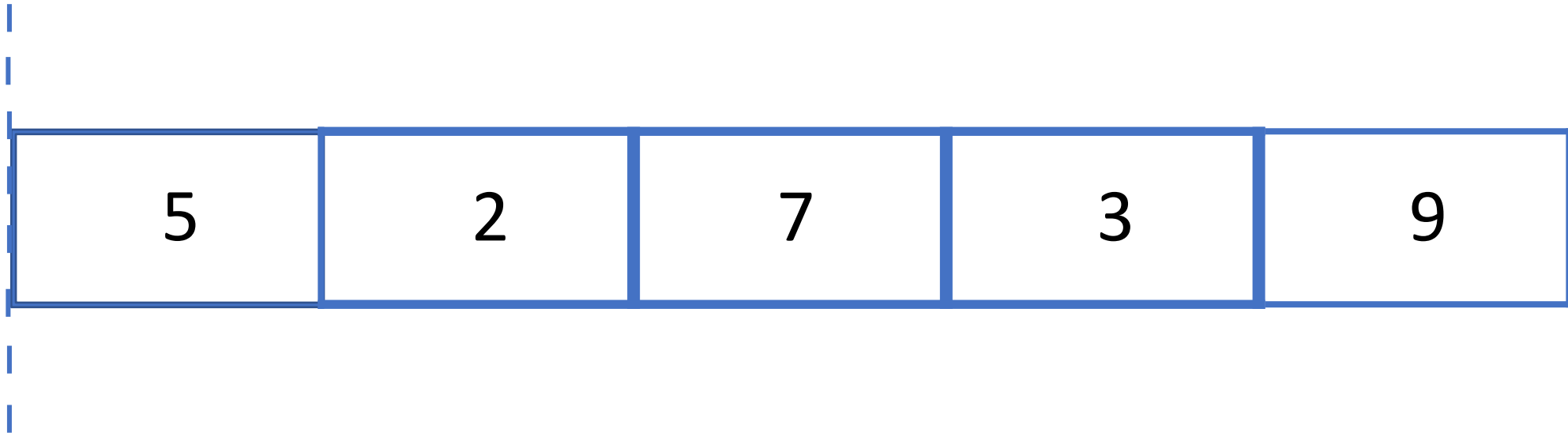
Selection Sort

1. Tenemos una secuencia desordenada
2. Buscar el menor dato 'x' en la secuencia
3. Intercambiar ese elemento 'x' con el primer elemento de la secuencia
4. Avanzar uno en la secuencia
5. Si aún queda secuencia, volver a 2

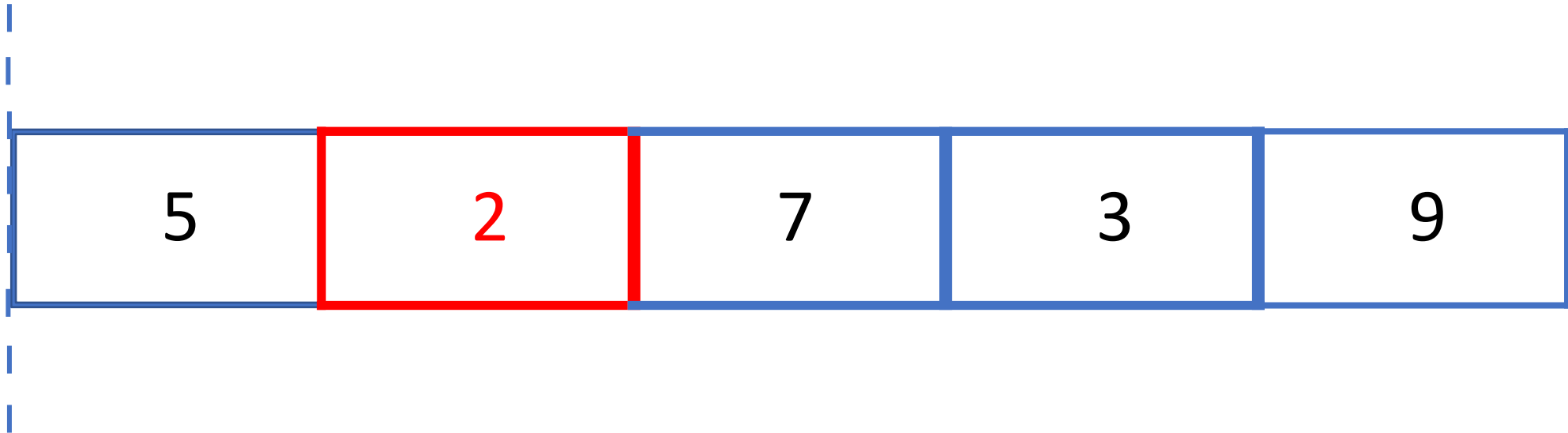
Veamos un
ejemplo...

5	2	7	3	9
---	---	---	---	---

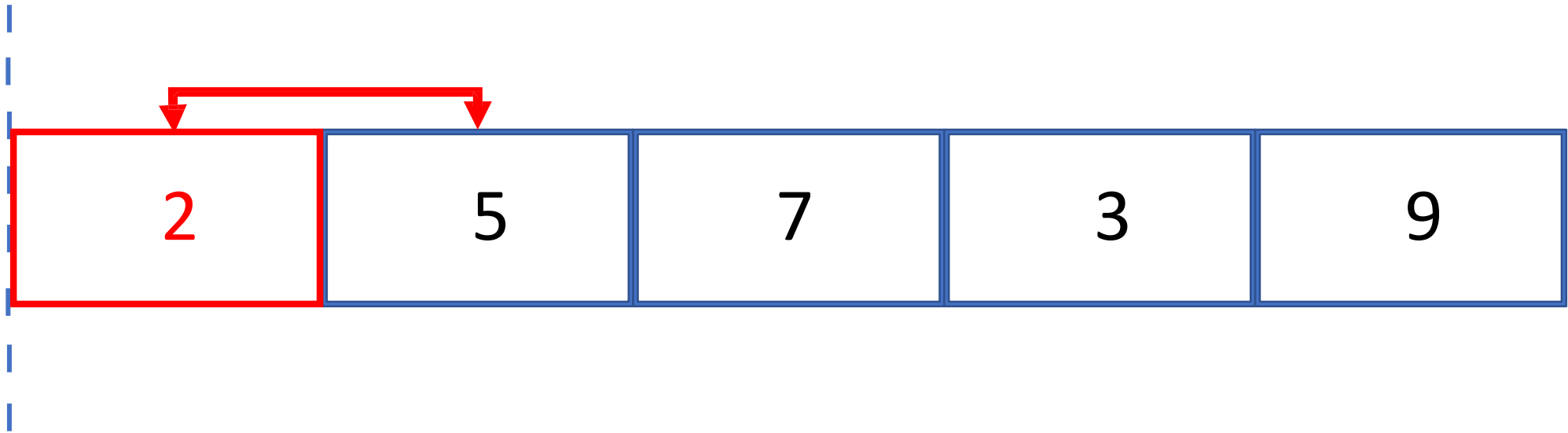
Partimos al principio



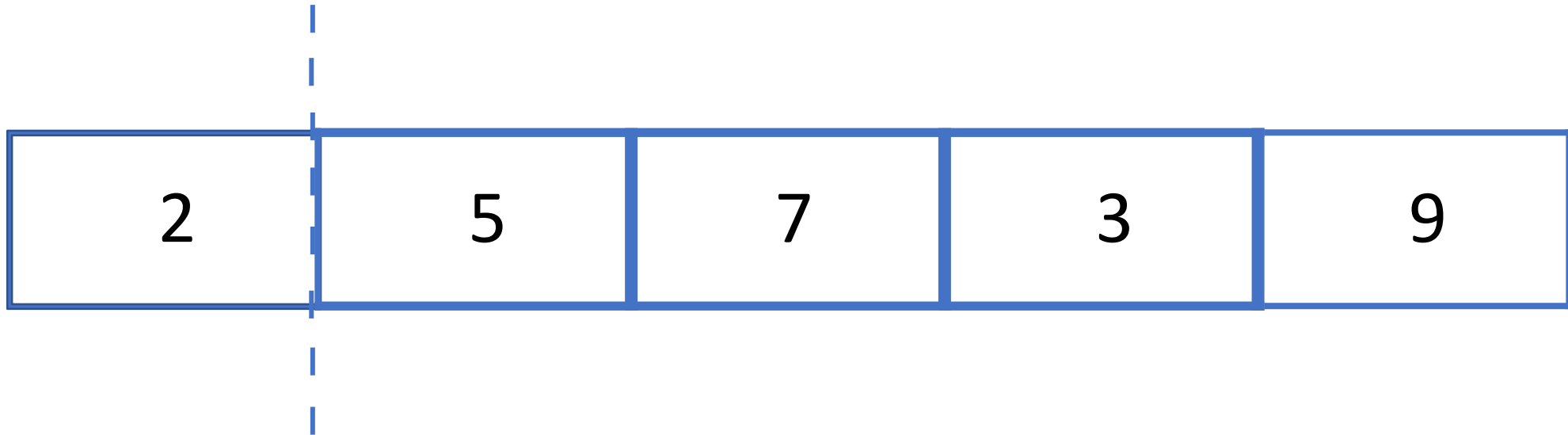
Seleccionamos el menor



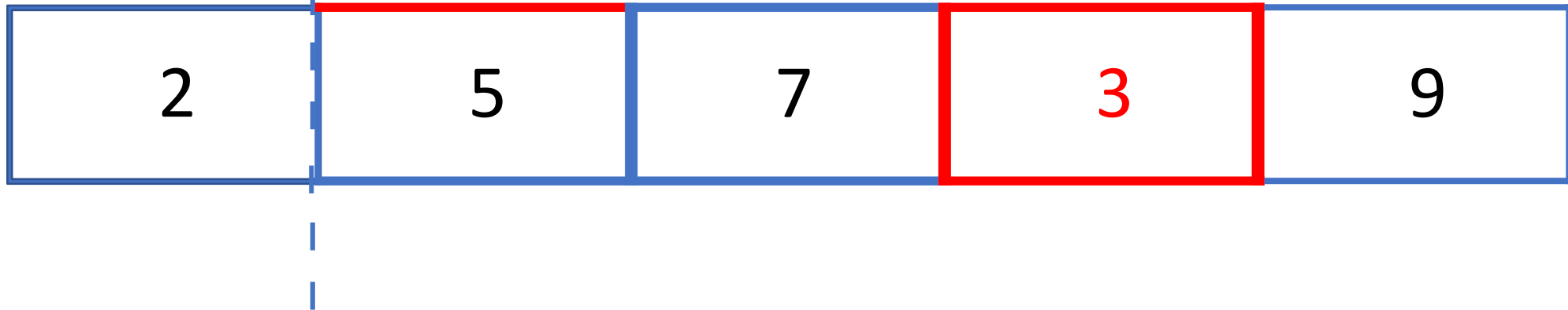
Intercambiamos a la primera posición



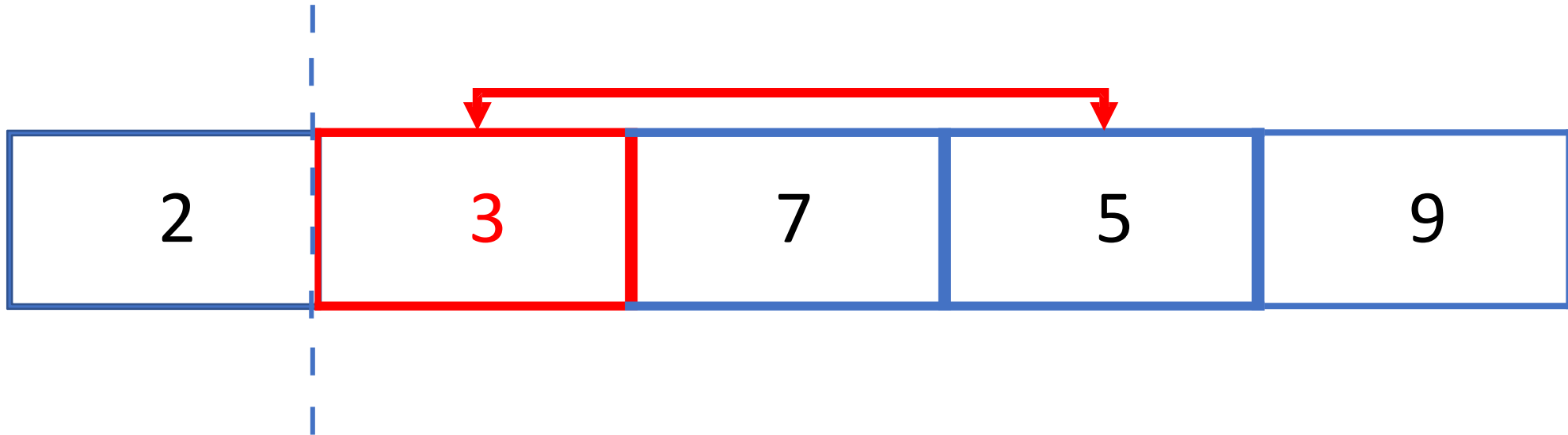
Nos movemos a la segunda posición



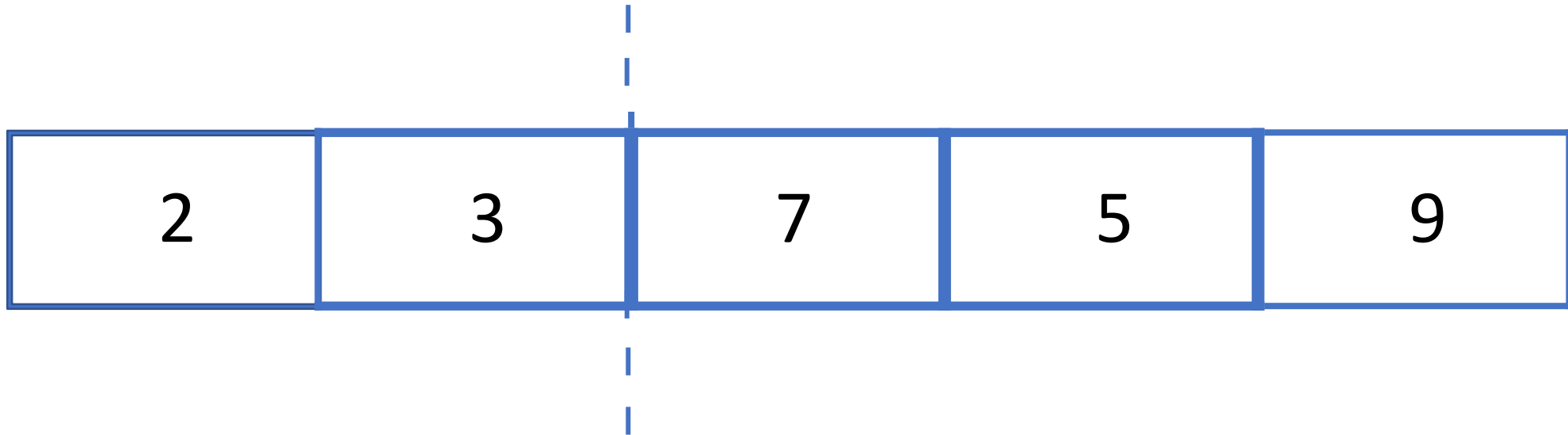
Seleccionamos el
menor...



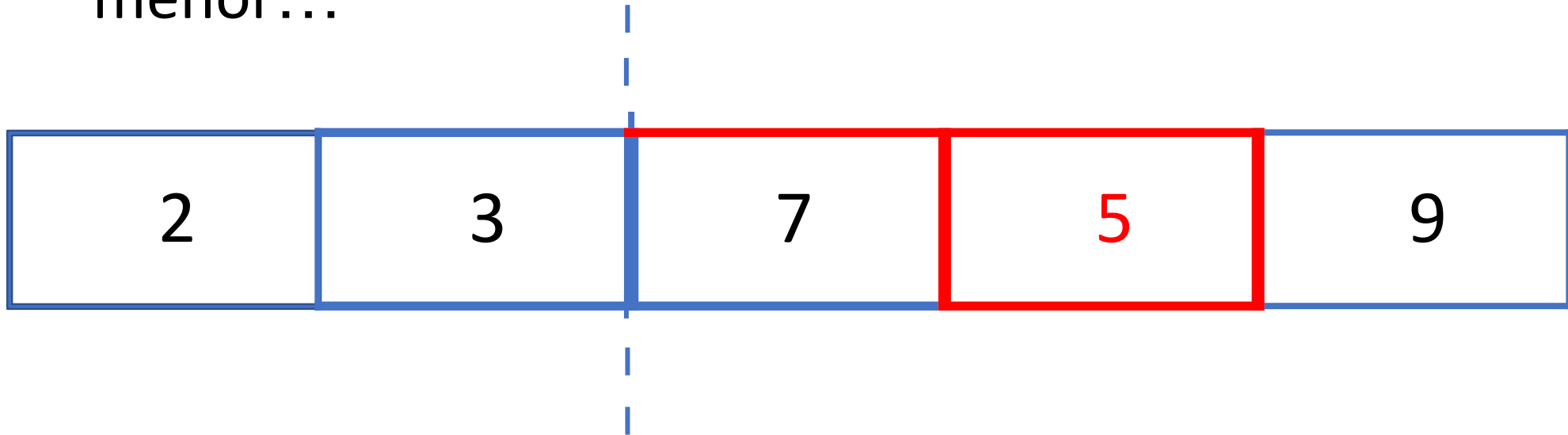
Intercambiamos con el primero



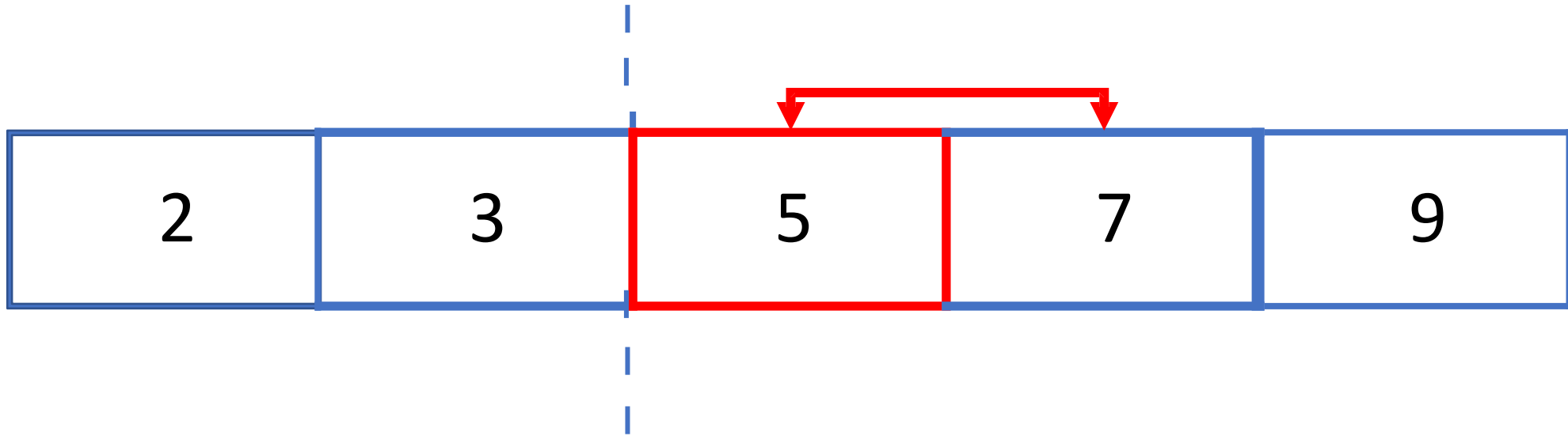
Nos movemos a la tercera posición



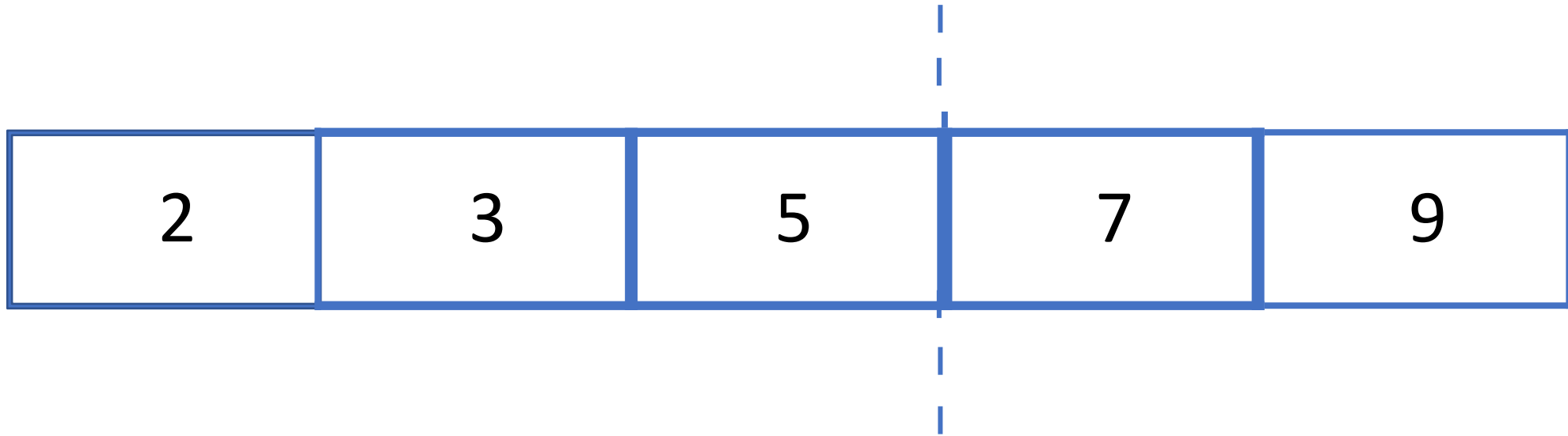
Seleccionamos el
menor...



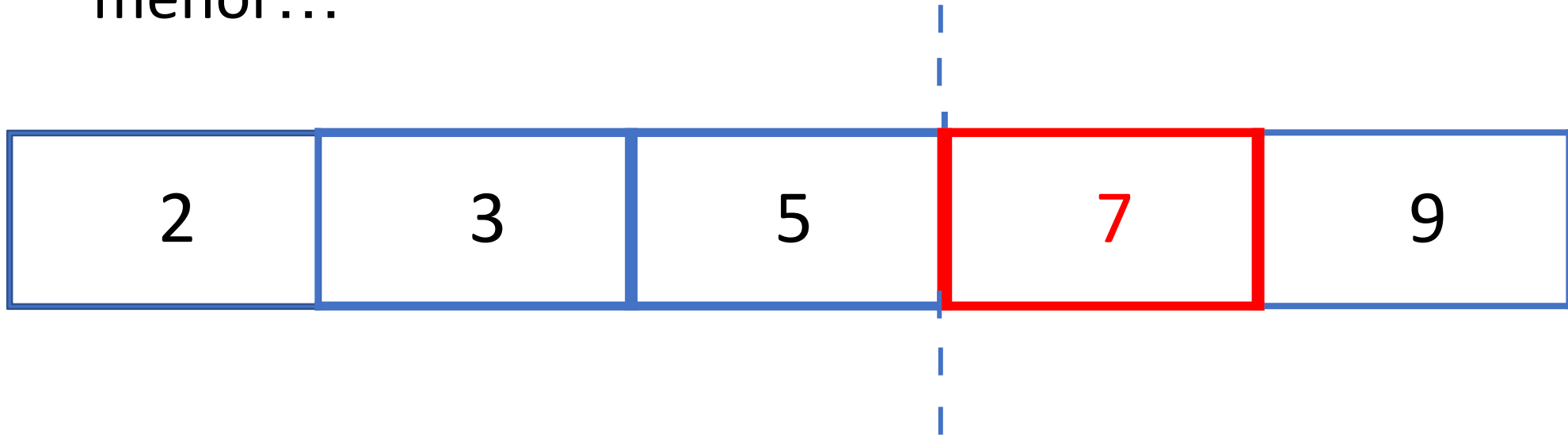
Intercambiamos con el primero



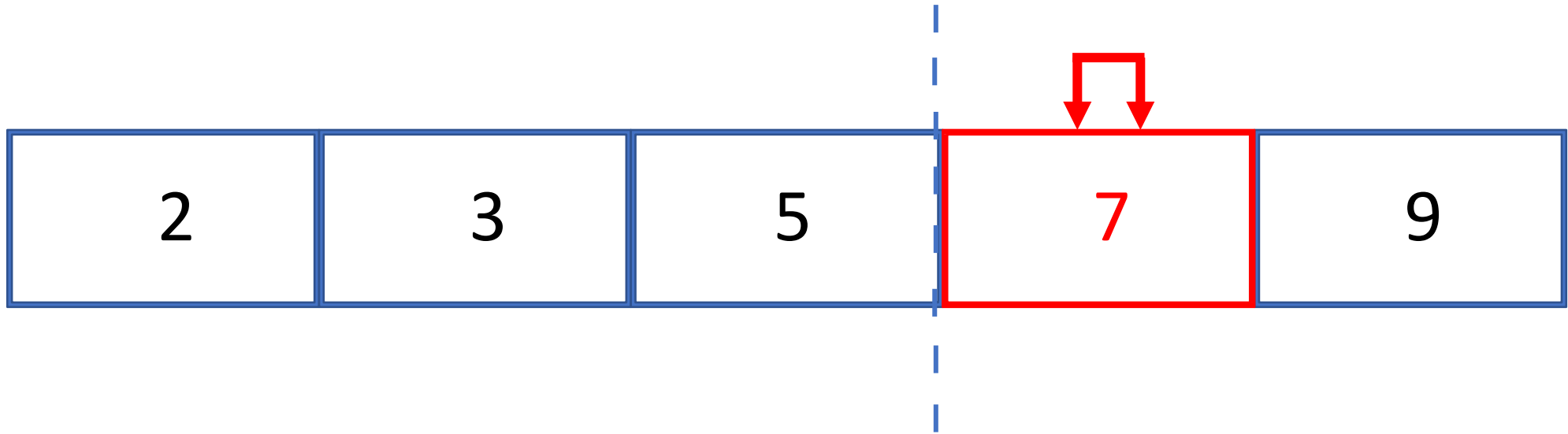
Nos movemos a la cuarta posición



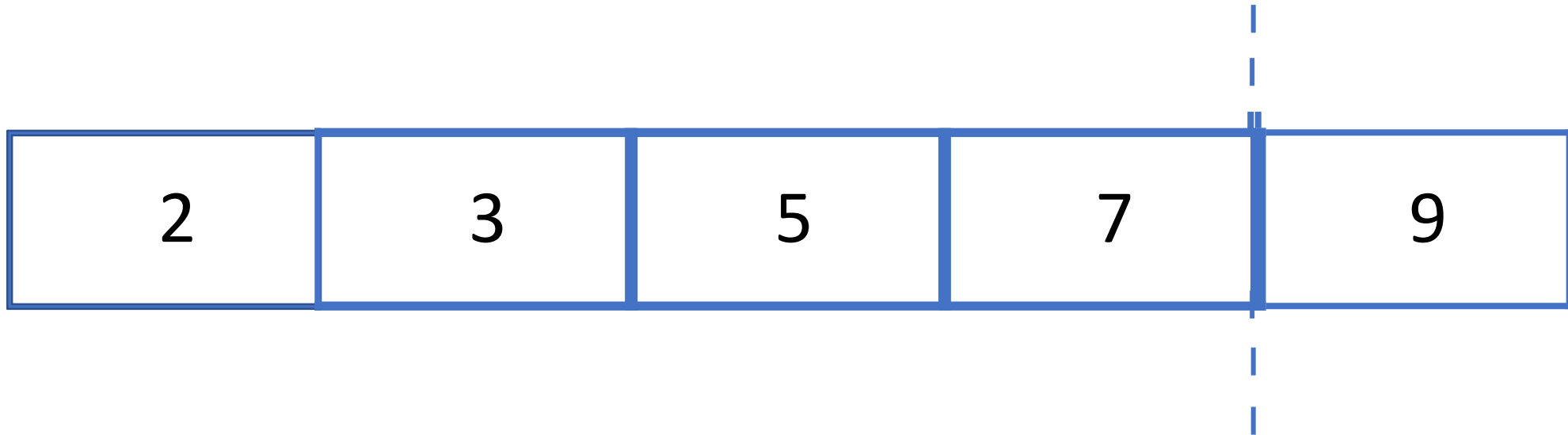
Seleccionamos el
menor...



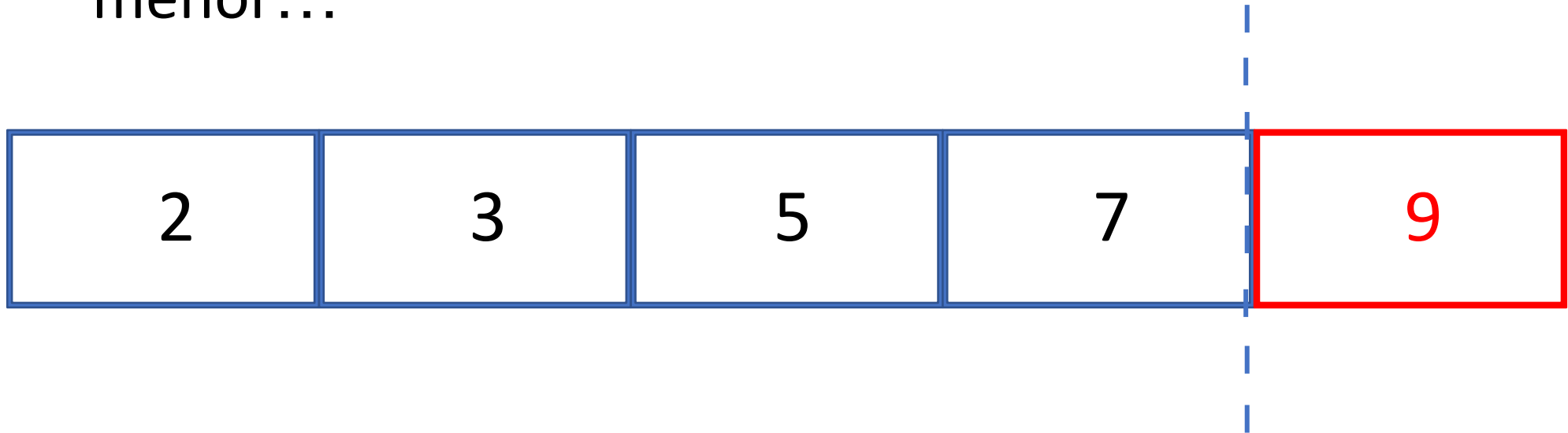
Intercambiamos con el primero



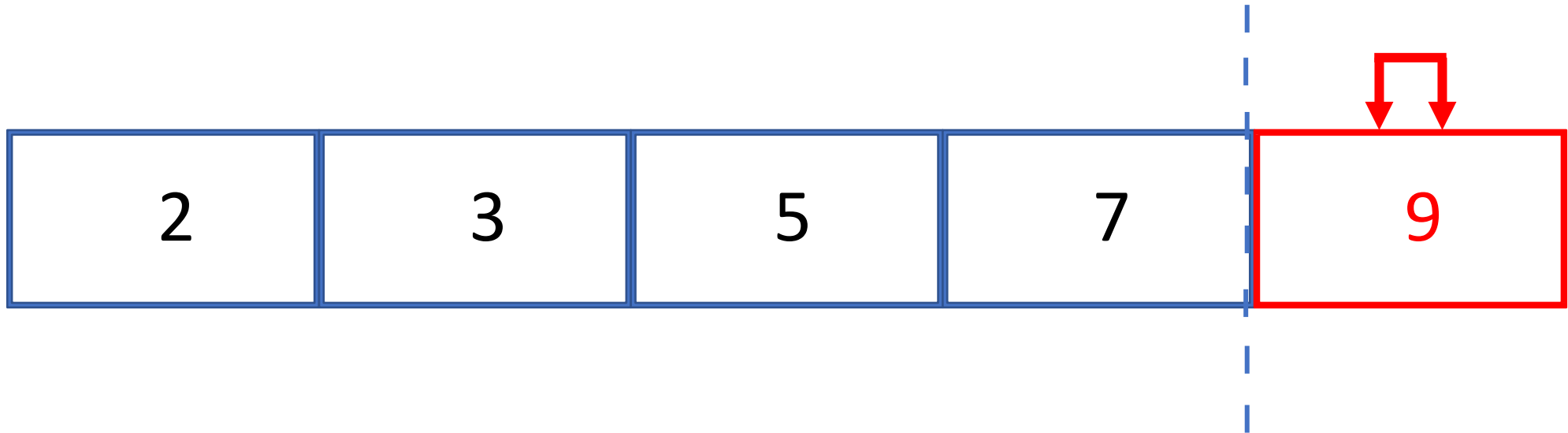
Nos movemos a la quinta posición



Seleccionamos el
menor...



Intercambiamos con el primero



Quedó ordenado!!

2	3	5	7	9
---	---	---	---	---

Insertion Sort

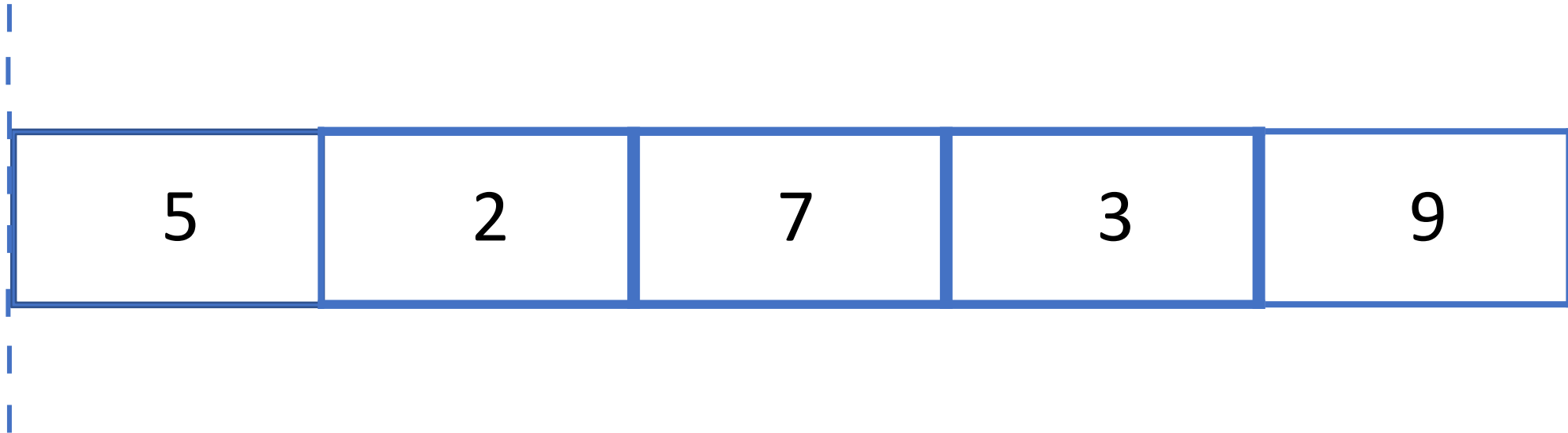
1. Tenemos una secuencia desordenada
2. Tomar el primer dato 'x' de la secuencia
3. Insertar 'x' en los elementos anteriores de manera que quede ordenado
4. Avanzar uno en la secuencia
5. Si quedan elementos en la secuencia, volver a 2



Veamos el mismo ejemplo
anterior...

5	2	7	3	9
---	---	---	---	---

Partimos al principio

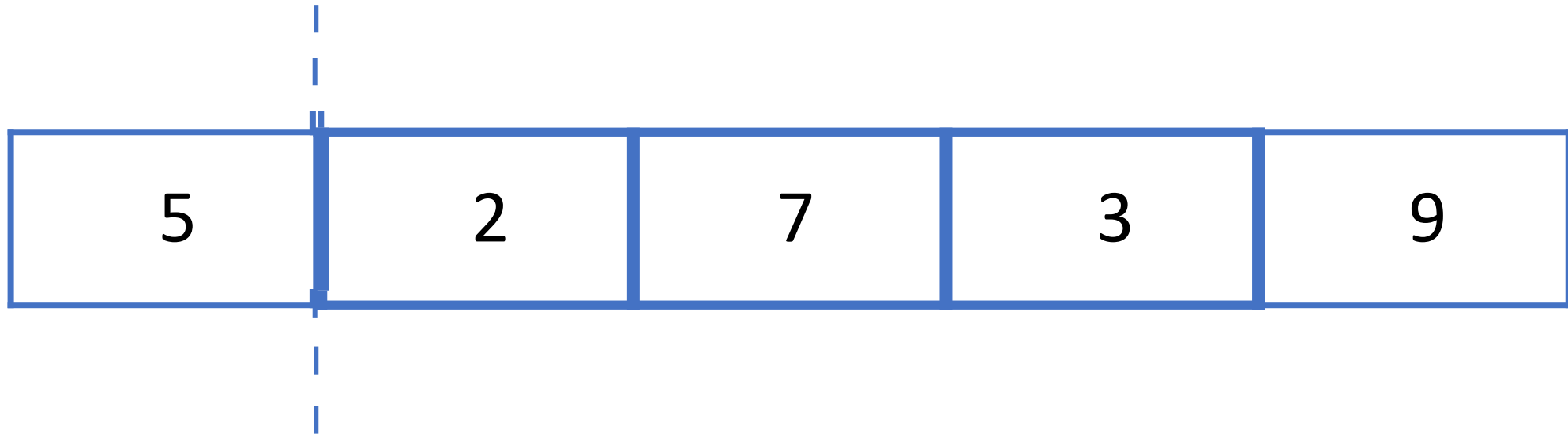


Comparamos el primero con el anterior

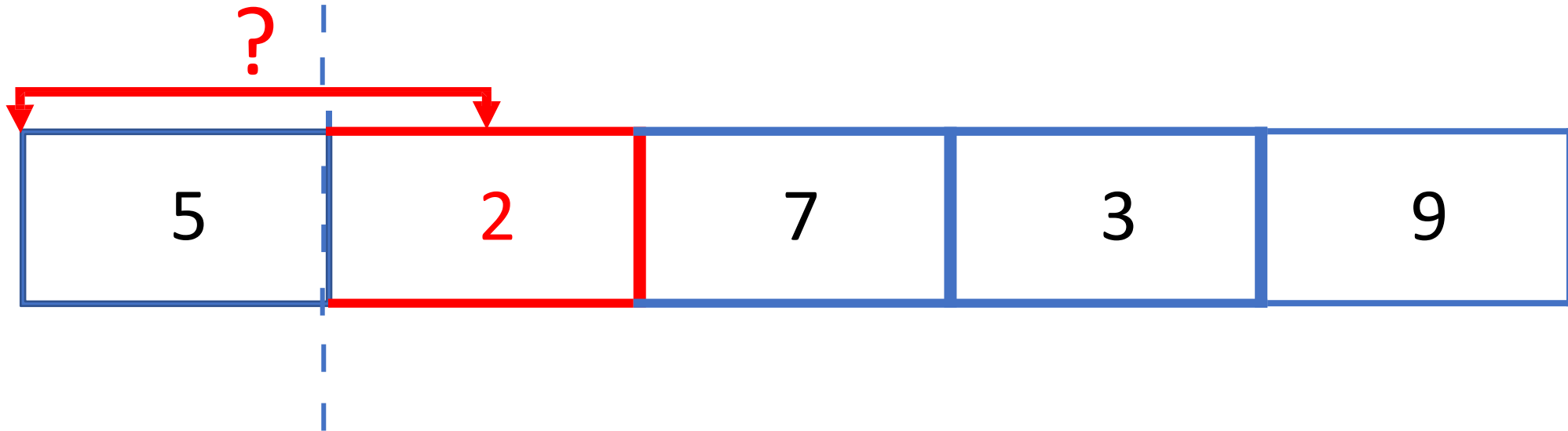
(en este caso no tenemos anterior, así que lo dejamos así)



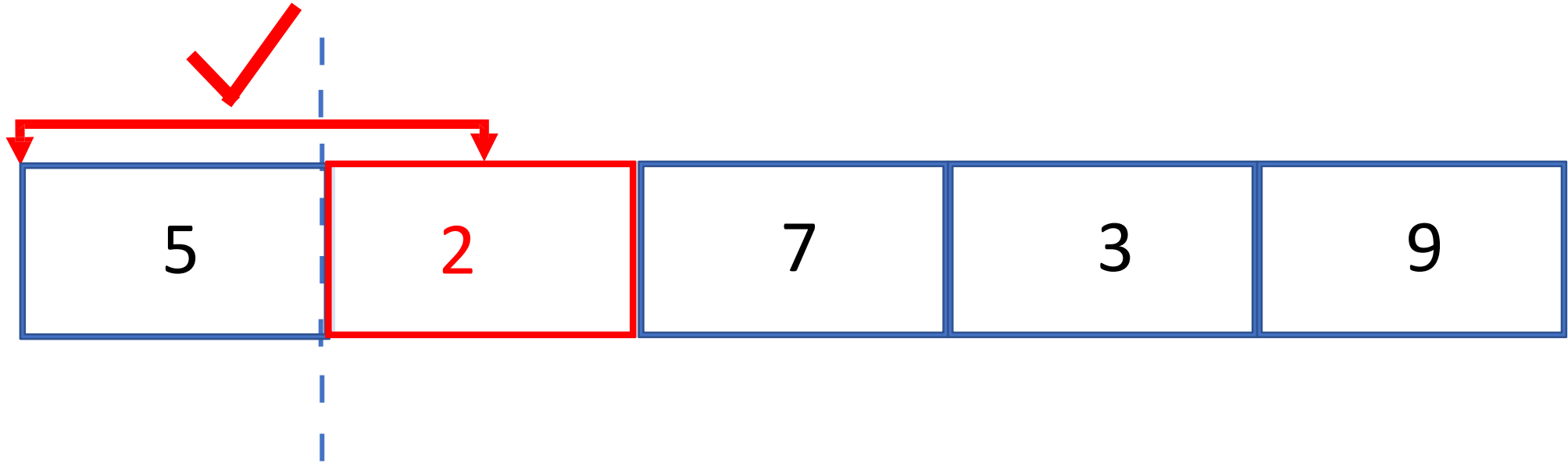
Nos movemos a la segunda posición



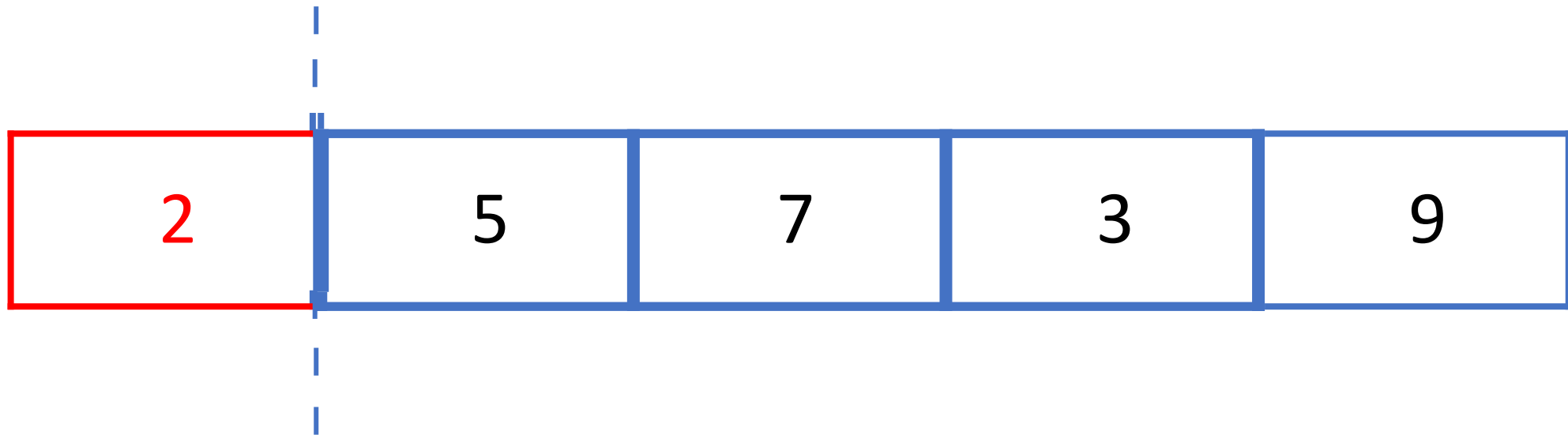
Vemos en qué posición debería quedar el primero



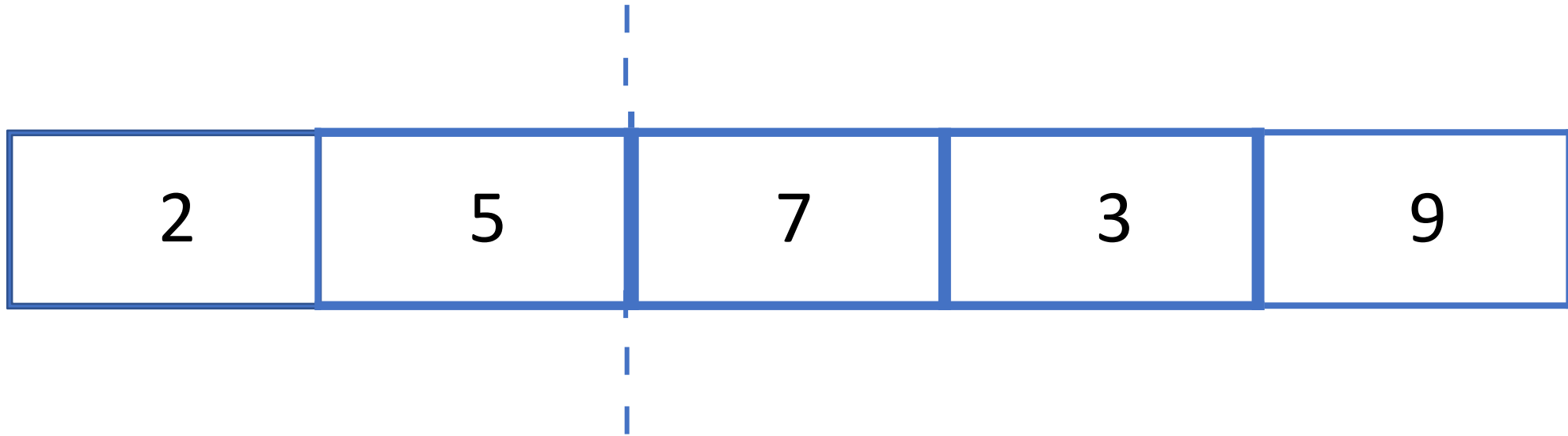
Como sabemos que va atrás, movemos la casilla.



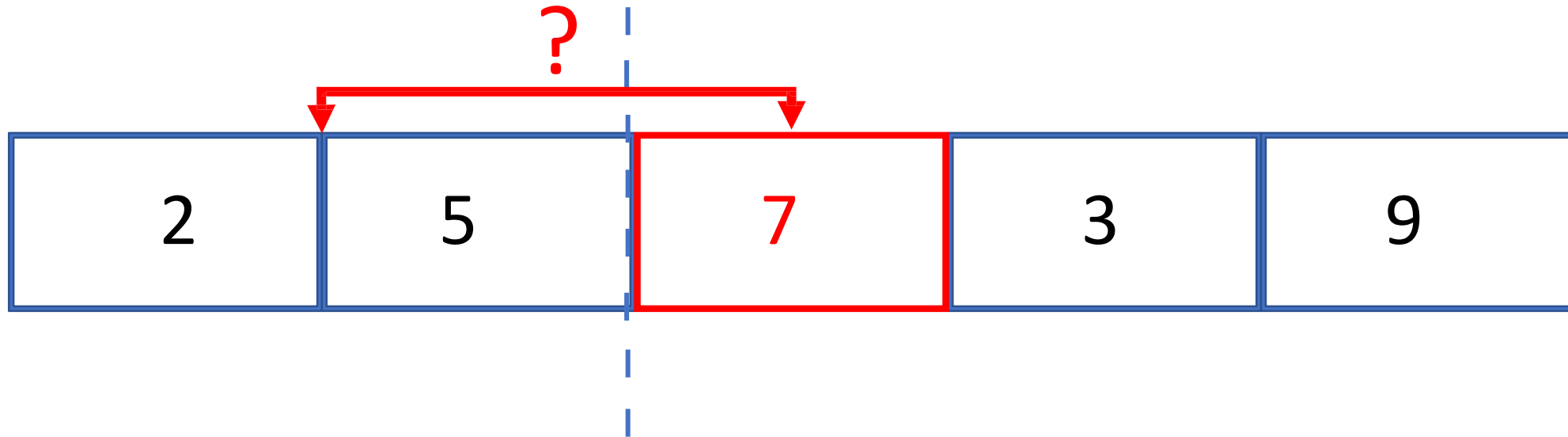
Reposicionamos



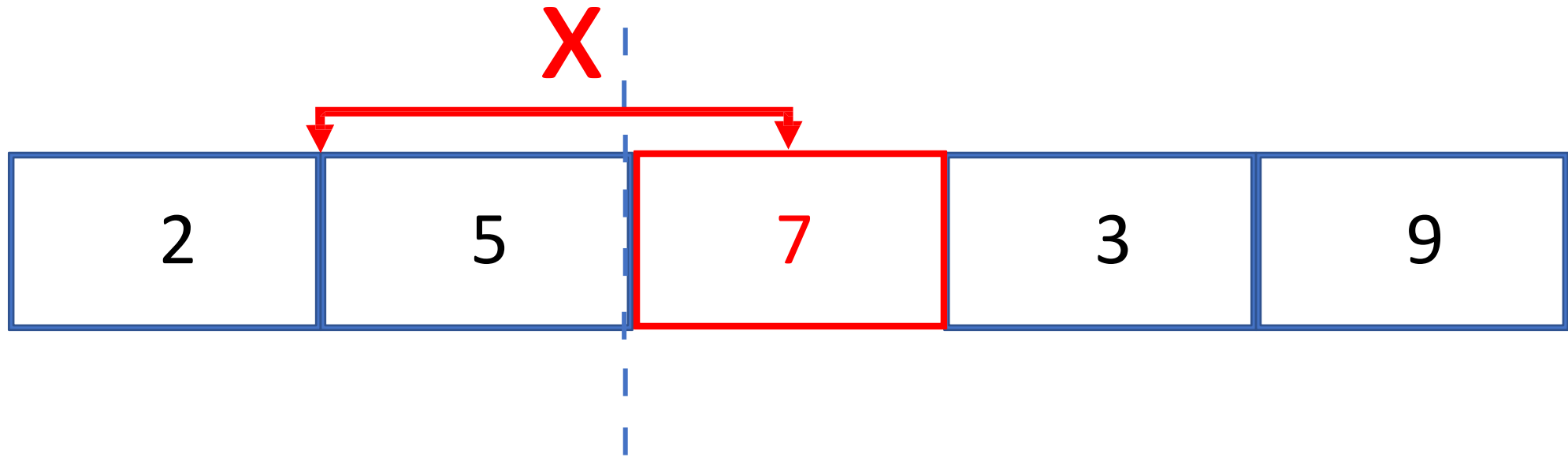
Nos movemos a la tercera posición



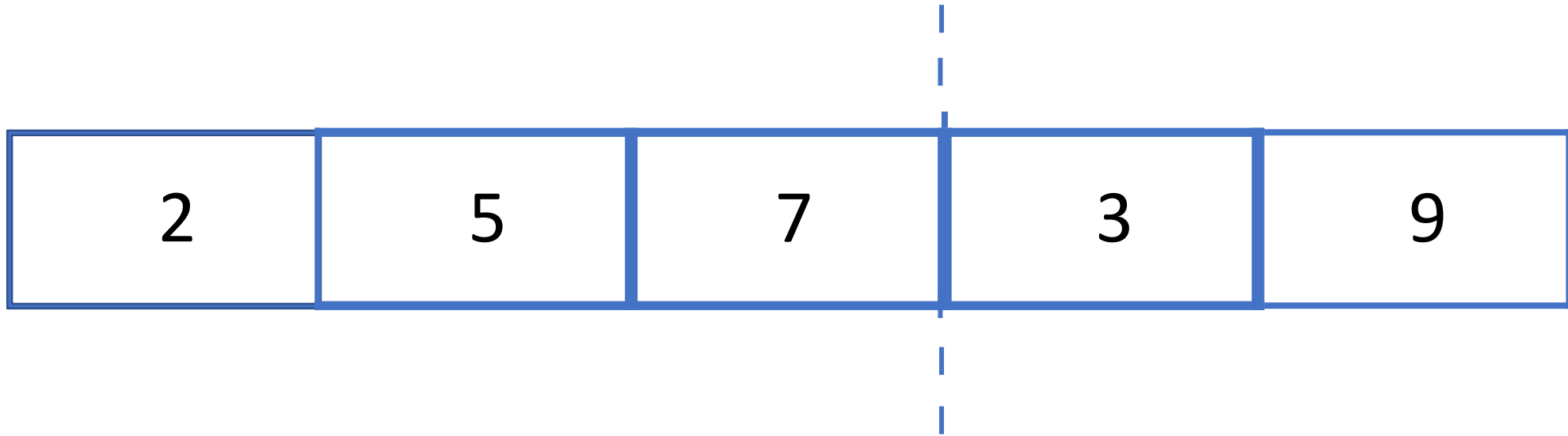
Vemos en qué posición debería quedar el primero



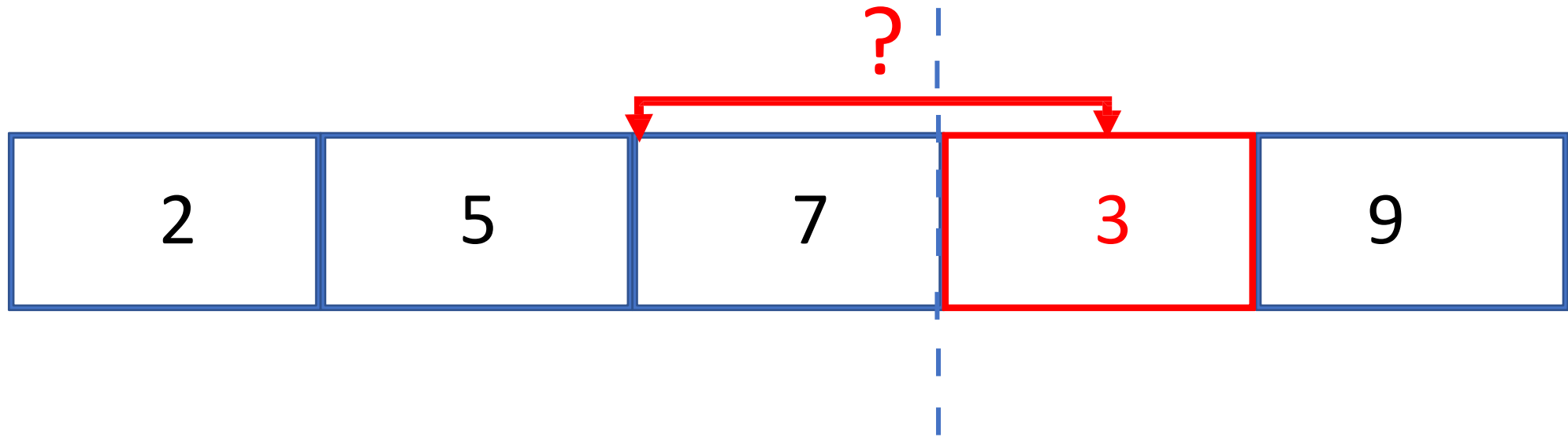
No pasa nada, lo dejamos así



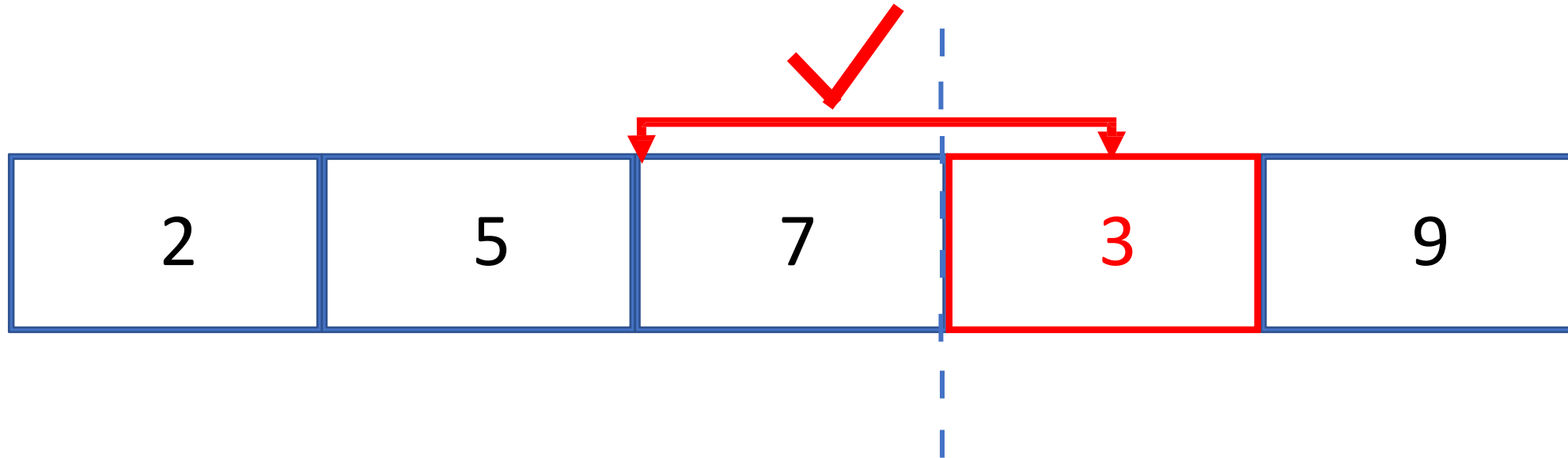
Nos movemos a la cuarta posición



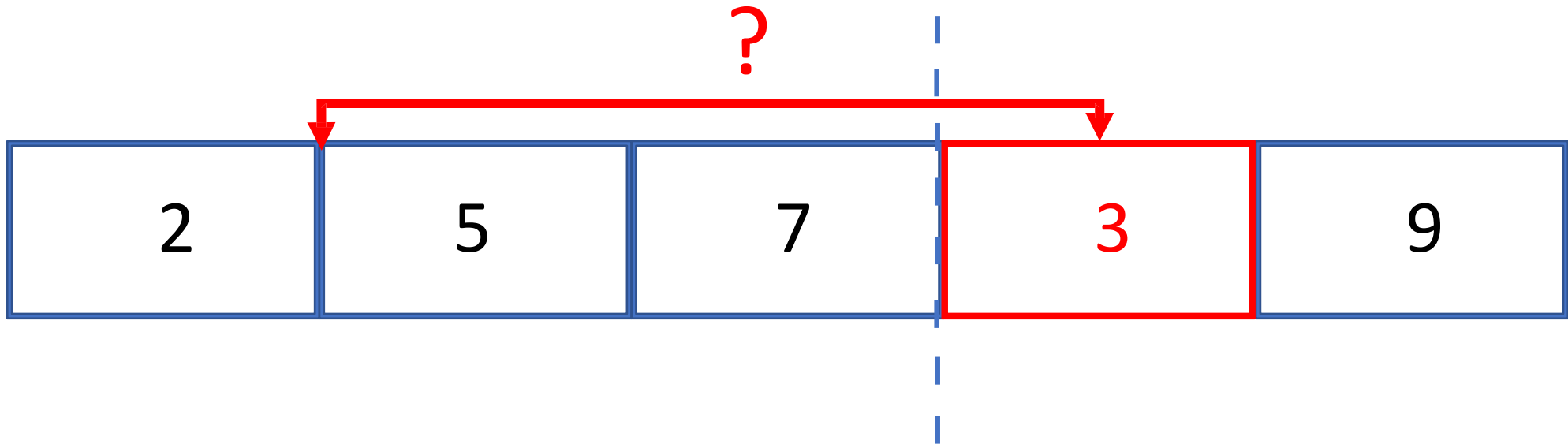
Vemos en qué posición debería quedar el primero



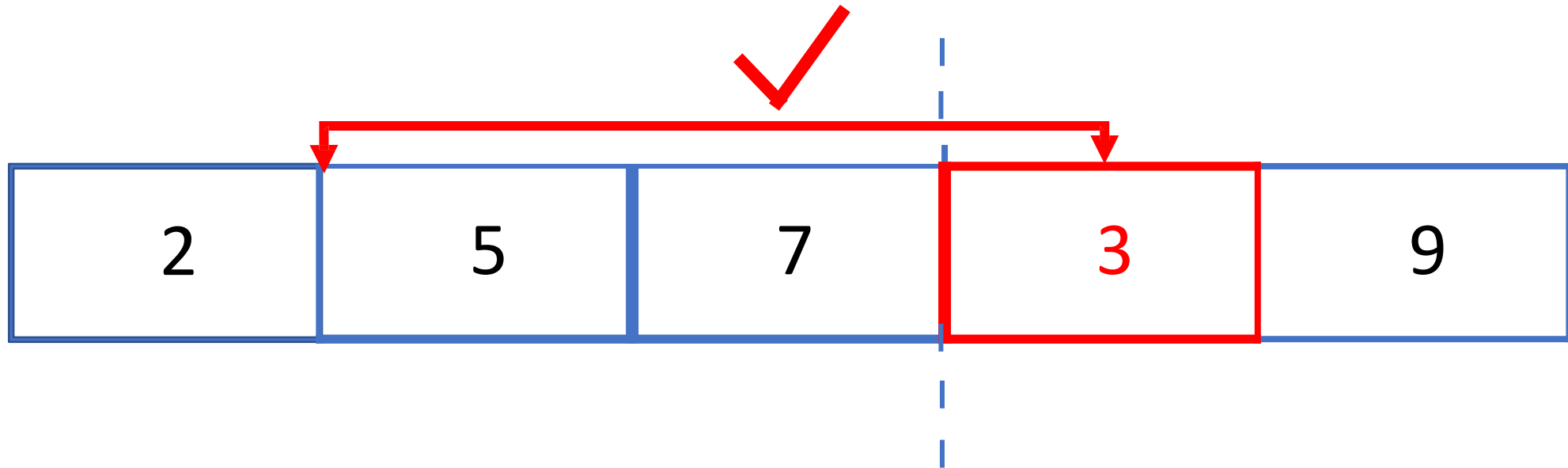
Sabemos que es menor, pero no necesariamente va ahí



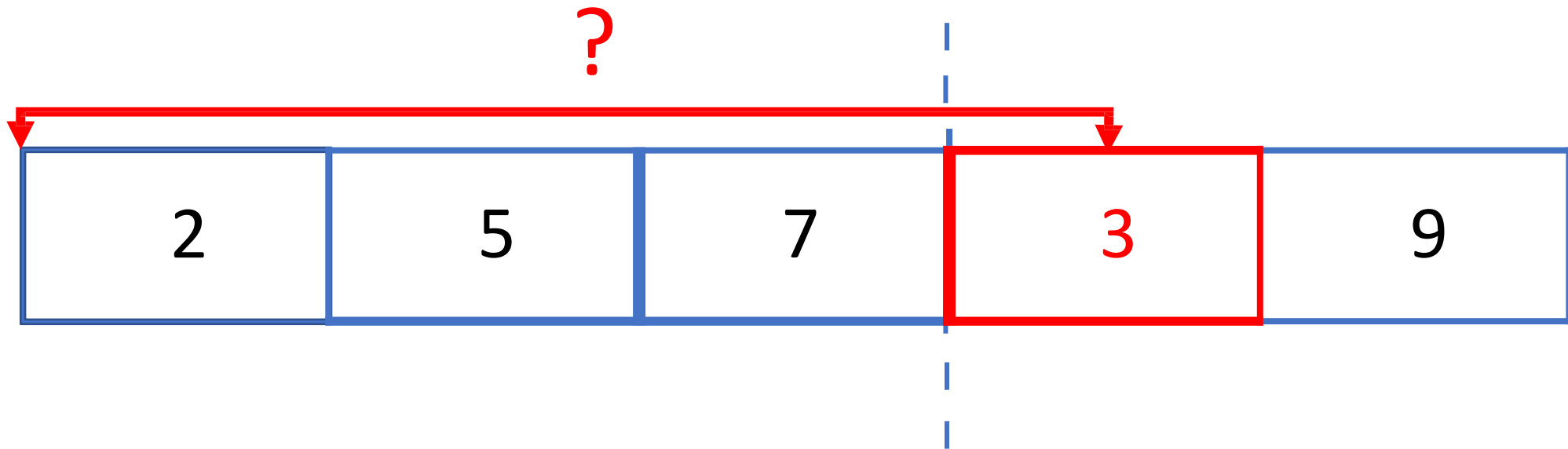
Vemos en otra posición



Sabemos que es menor, pero no necesariamente va ahí

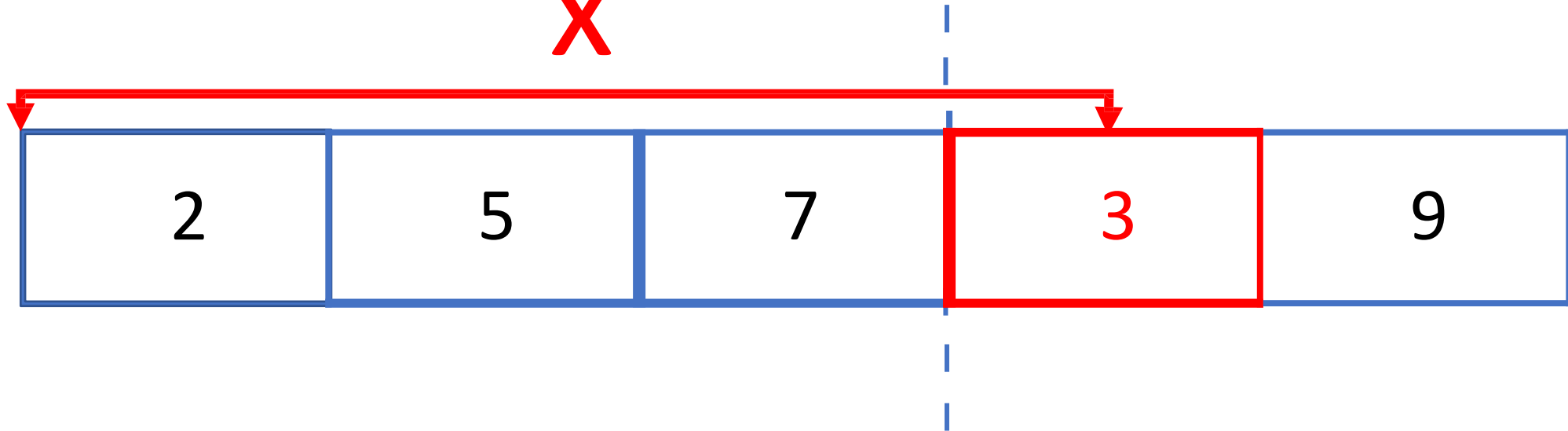


Sabemos que es menor, pero no necesariamente va ahí

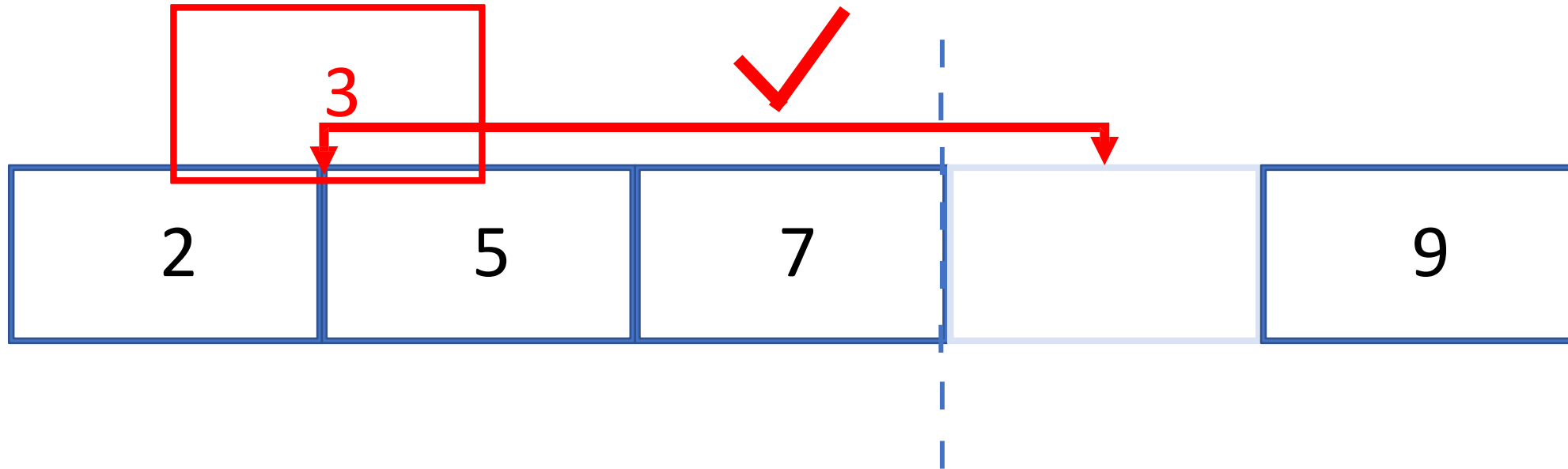


Entonces era el de antes

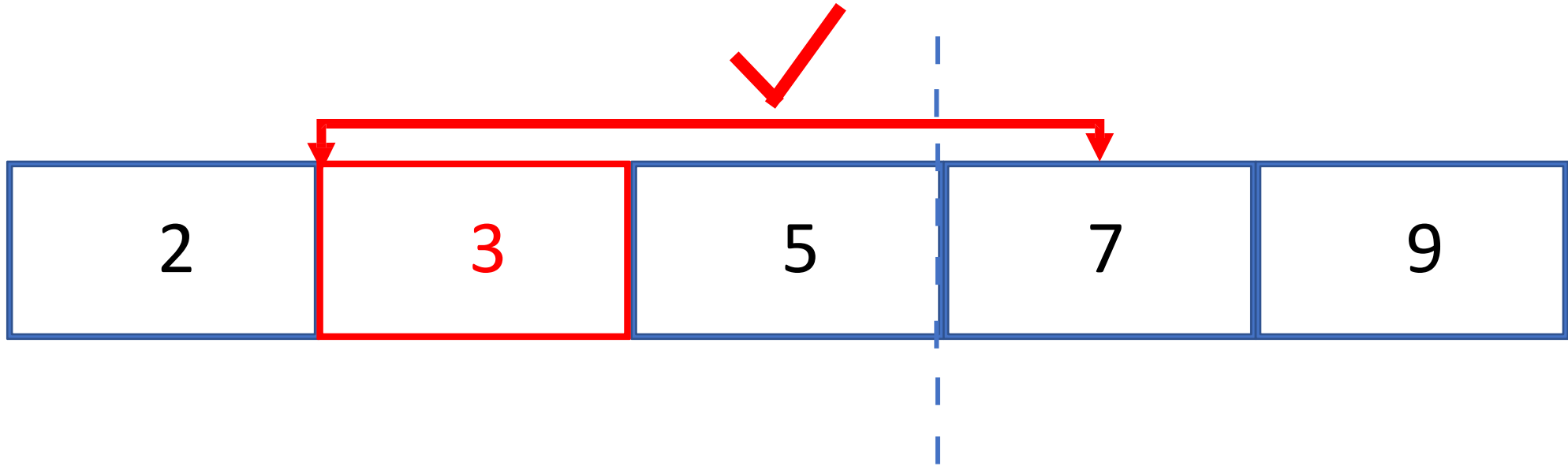
X



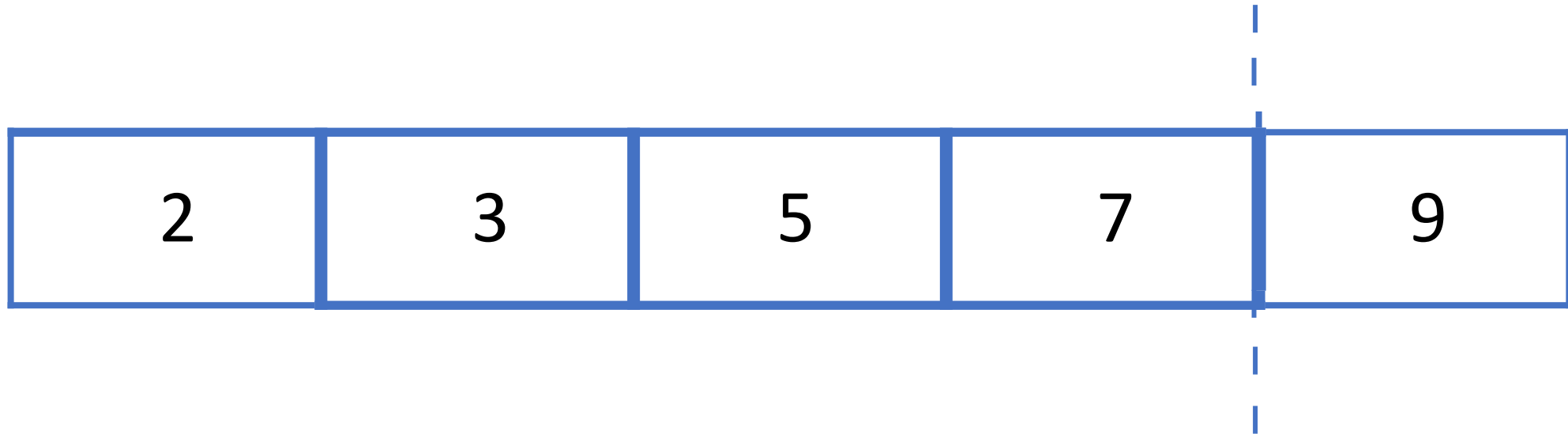
Colocamos donde corresponde y movemos los otros.



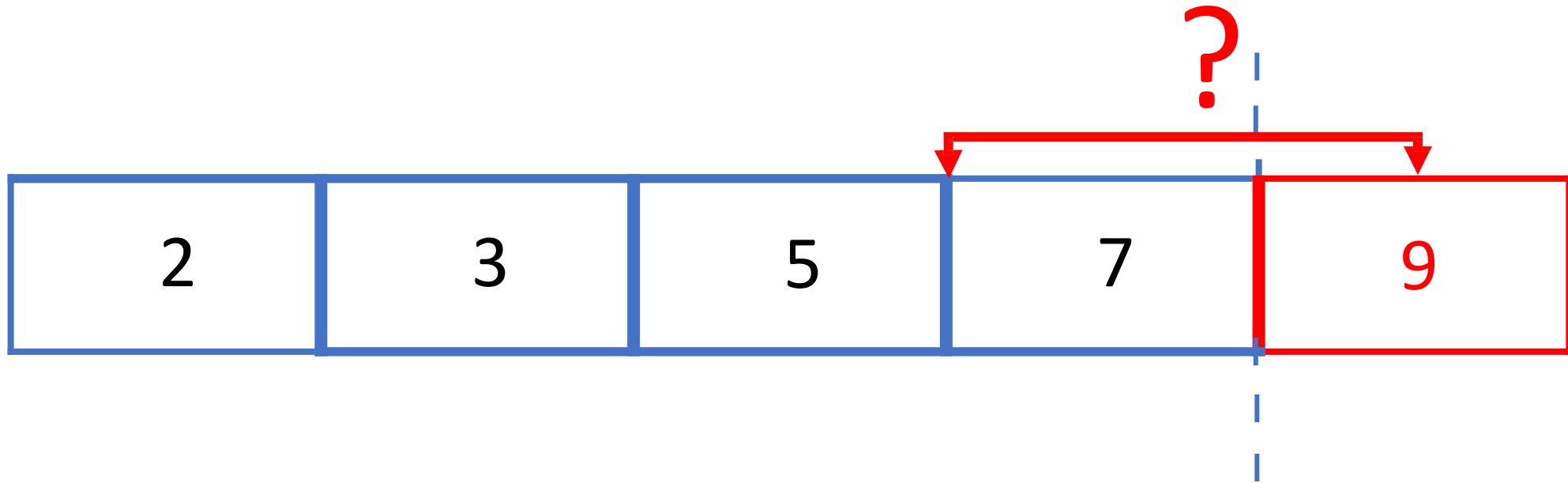
Sabemos que es menor, pero no necesariamente va ahí



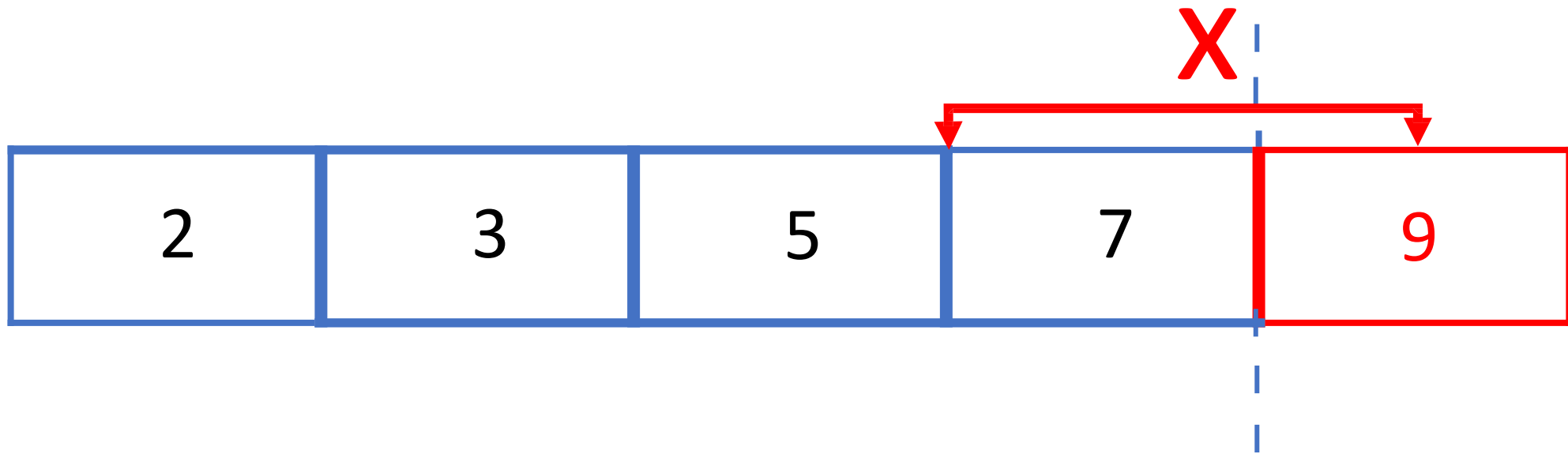
Nos movemos a la quinta posición



Vemos en qué posición debería quedar el primero



No pasa nada, lo dejamos así



Esta ordenado!!

2	3	5	7	9
---	---	---	---	---

Gnome Sort

1. Tenemos una secuencia desordenada de datos
2. Se va comparando cada par de la secuencia de izquierda a derecha
3. Una vez se encuentra un par desordenado, con dato 'x' mal posicionado
4. Retrocede comparando en pares dejando cada dato ordenado
5. Avanza nuevamente desde un principio comparando de a pares
6. Si quedan elementos desordenados, vuelve a 2



1



4	9	8	5
---	---	---	---

2



4	9	8	5
---	---	---	---

3



4	8	9	5
---	---	---	---

4



4	8	9	5
---	---	---	---

5



4	8	9	5
---	---	---	---

6



4	8	5	9
---	---	---	---

7



4	5	8	9
---	---	---	---

8



4	5	8	9
---	---	---	---

9



4	5	8	9
---	---	---	---

10



4	5	8	9
---	---	---	---

DONE

<https://www.youtube.com/watch?v=kPRA0W1kECg>