Ayudantía 1

Inducción
Notación Asintótica
Correctitud
Introducción a Sorting

Inducción

Principio de Inducción Simple (PIS)

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

$$P(n_0) \text{ es verdadero } \ n_0 \in N \text{ cumple con propiedad P}$$

$$\text{Si } P(n), \text{entonces } P(n+1)$$
$$\text{cada vez que n cumple con la propiedad n+1 también la cumple}$$

$$\text{Entonces todos los elementos de N a partir de } n_0 \text{ cumplen con la propiedad}$$

Principio de Inducción Simple (PIS)

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

Base Inductiva:

$$P(n_0) \text{ es verdadero } \; n_0 \in N \text{ cumple con propiedad P}$$

Paso Inductivo:

$$\text{Si } P(n), \text{entonces } P(n+1)$$

$$\text{cada vez que n cumple con la propiedad n+1 también la cumple}$$

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

Paso inductivo

Si $P(n)$ → **Hipótesis inductiva** (Es lo que asumimos)

entonces $P(n+1)$→ **Tesis Inductiva** (Es lo que debemos llegar)

Importante: **Nunca** partir desde la Tesis Inductiva y llegar a la Hipótesis Inductiva

Ejemplo Inducción:

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

1. Base Inductiva: Para **n** = 1:
2. Hipótesis Inductiva: Asumimos que la afirmación se cumple para un número **N**.
3. Tesis Inductiva: Demostramos que la afirmación se cumple para **N + 1.**

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

1. Base Inductiva: Para **n** = 1:

$1^3 - 1 = 0 = 0 * 6$ 6 divide a $1^3 - 1$

2. Hipótesis Inductiva: Asumimos que la afirmación se cumple para un número **N**.

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

3. Tesis Inductiva: Demostramos que la afirmación se cumple para **N + 1.**

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

Ejemplo Inducción:

Principio de Inducción por curso de valores (PICV) (o inducción fuerte)

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

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

Entonces P es verdadero para todos los elementos de N.

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

Ejemplo Inducción:

Por demostrar: Todo número natural n>1 puede ser escrito como una multiplicación de uno o más números primos.

1. **BI:**

2. **HI:**

3. **TI:**

Ejemplo Inducción:

Por demostrar: Todo número natural n>1 puede ser escrito como una multiplicación de uno o más números primos.

1. **BI:** Para n=2, 2 es un número primo.

2. **HI:** Supongamos que la propiedad se cumple para todo $k \in N$, tal que $k < n$, con un n arbitrario.

3. **TI:** -Si n es primo, solución trivial.

   -Si n no es primo…

Notación Asintótica

¿Para qué sirve?

- Determinar tiempo de respuesta de nuestro algoritmo (runtime)
- Determinar recursos computacionales
- Ver la escalabilidad de nuestra función

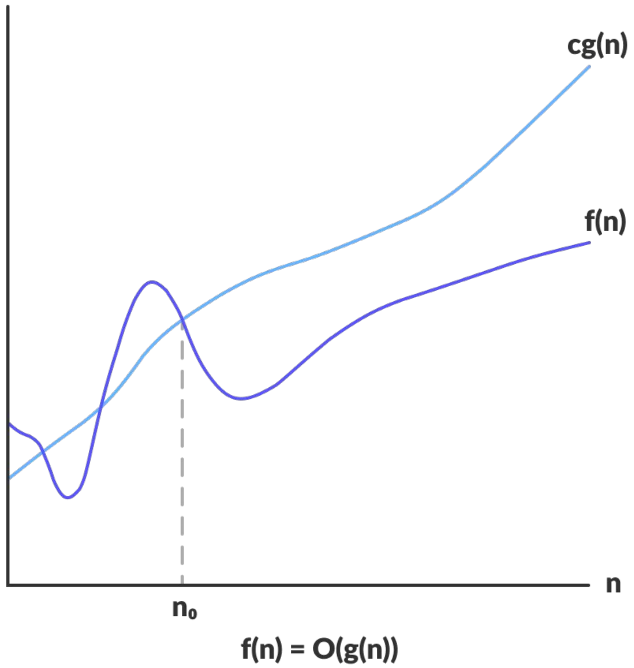
Nos permite elegir algoritmos más eficientes. Sin embargo, muchas veces el análisis no es trivial

Notación O

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

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : f(n) \leq cg(n), \forall n > n_0\}$$

Notación O

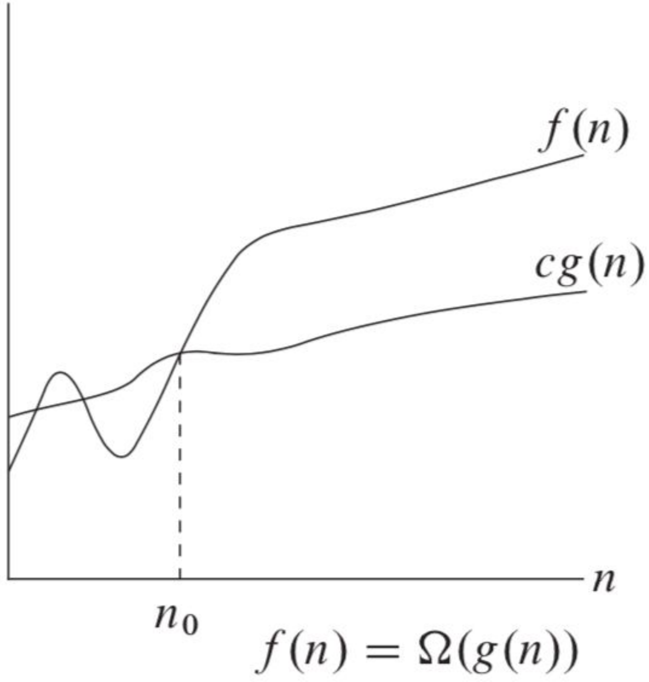


Notación Omega

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

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^{+} | \exists c \in \mathbb{R}^{+}, \exists n_0 \in \mathbb{N} : 0 < cg(n) \leq f(n), \forall n > n_0\}$$

Notación Omega

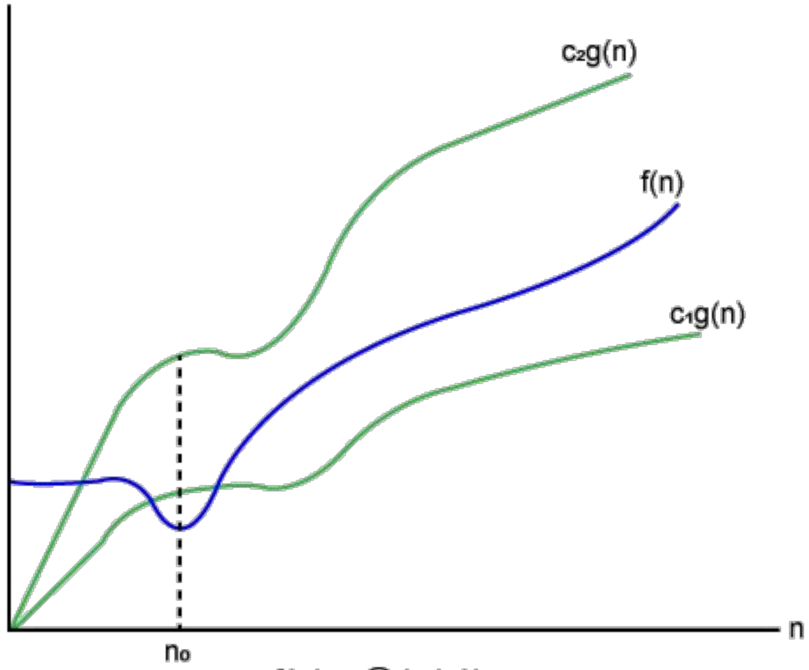


Notación Theta

Diremos que f (n) ϵ $\boldsymbol{\Theta}(g$(n)) si f (n) ϵ $\Omega(g$(n)) y f (n) ϵ$O(g$(n)), es decir:

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R}^+ | \exists c_1, c_2 \in \mathbb{R}^+ : 0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n > n_0\}$$

Notación Theta



$f(n) = \Theta(g(n))$

Órdenes de complejidad

$T(n) = c$ $\mathcal{O}(1) \rightarrow$ Orden constante.

$T(n) = log_2 n$ $\mathcal{O}(logn) \rightarrow$ Orden logarítmico.

$T(n) = an + b$ $\mathcal{O}(n) \rightarrow$ Orden lineal.

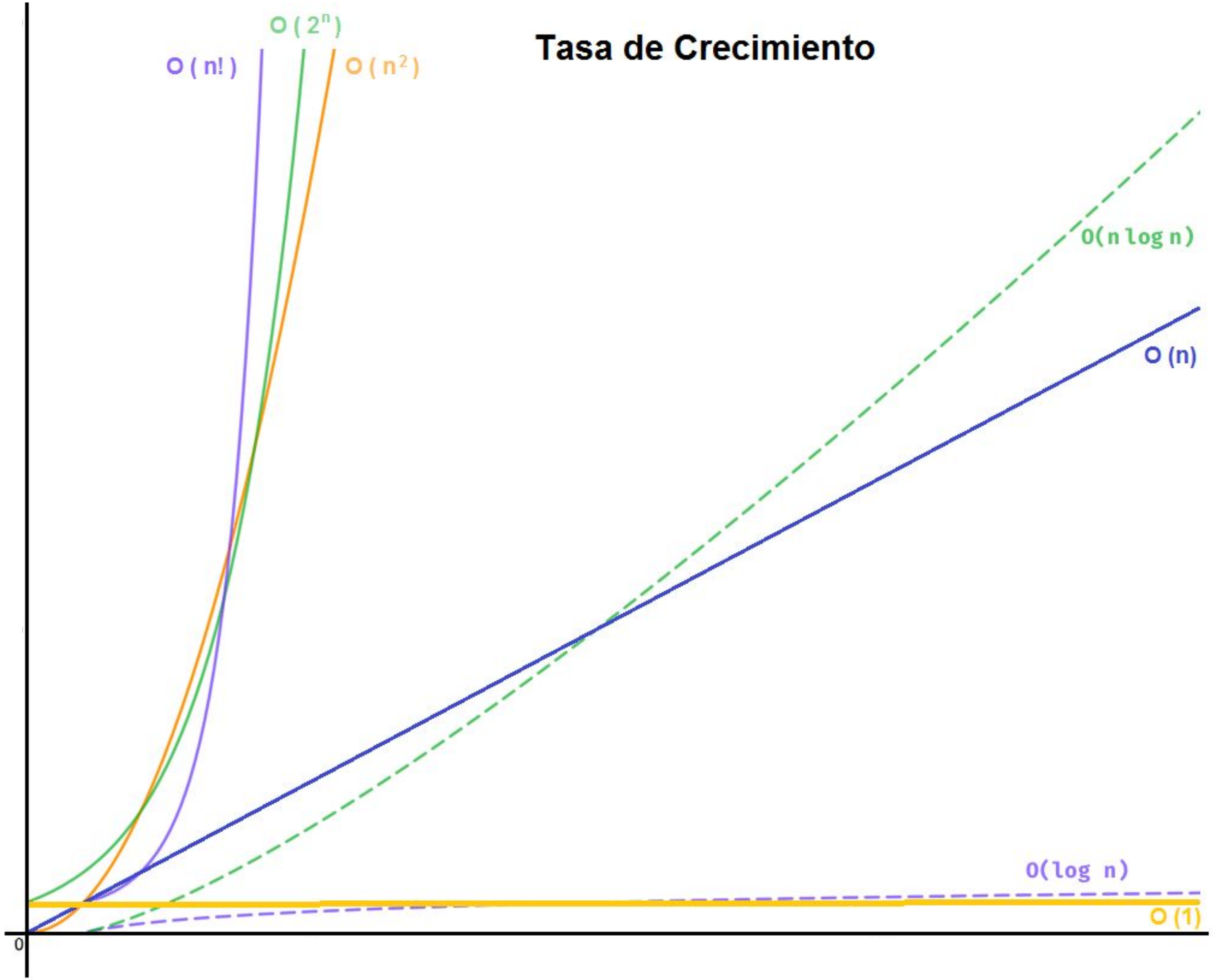
$T(n) = nlog_2 n$ $\mathcal{O}(nlogn) \rightarrow$ Orden n logn.

$T(n) = a_0 n^m + a_1 n^{m-1} \ldots a_{m-1} n + c$ $\mathcal{O}(n^m) \rightarrow$ Orden polinomial.

$T(n) = c^n$ $\mathcal{O}(2^n) \rightarrow$ Orden exponencial.

$T(n) = n!$ $\mathcal{O}(n!) \rightarrow$ Orden factorial.

Tasa de Crecimiento



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.

Correctitud

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.

2. Se repite el paso 1. hasta que se realice una iteración sin ningún intercambio.

Ejemplo Correctitud: Bubble Sort

1° Iteración

2	1	8	6

1	2	8	6

1	2	8	6

1	2	6	8

Ejemplo Correctitud: Bubble Sort

2° 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 removió 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

¿Cuál es el mejor caso?

¿Cuál es el peor caso?

Selection Sort

1. Tenemos una secuencia **desordenada**
2. Iniciar en **posición** 0
3. Buscar el **menor** dato ‘x’ en la secuencia
4. **Intercambiar** ese elemento ‘x’ con el elemento **actual** de la secuencia
5. **Avanzar** uno en la secuencia
6. **Si** aún queda secuencia, volver a 2

Selection Sort

Veamos un ejemplo…

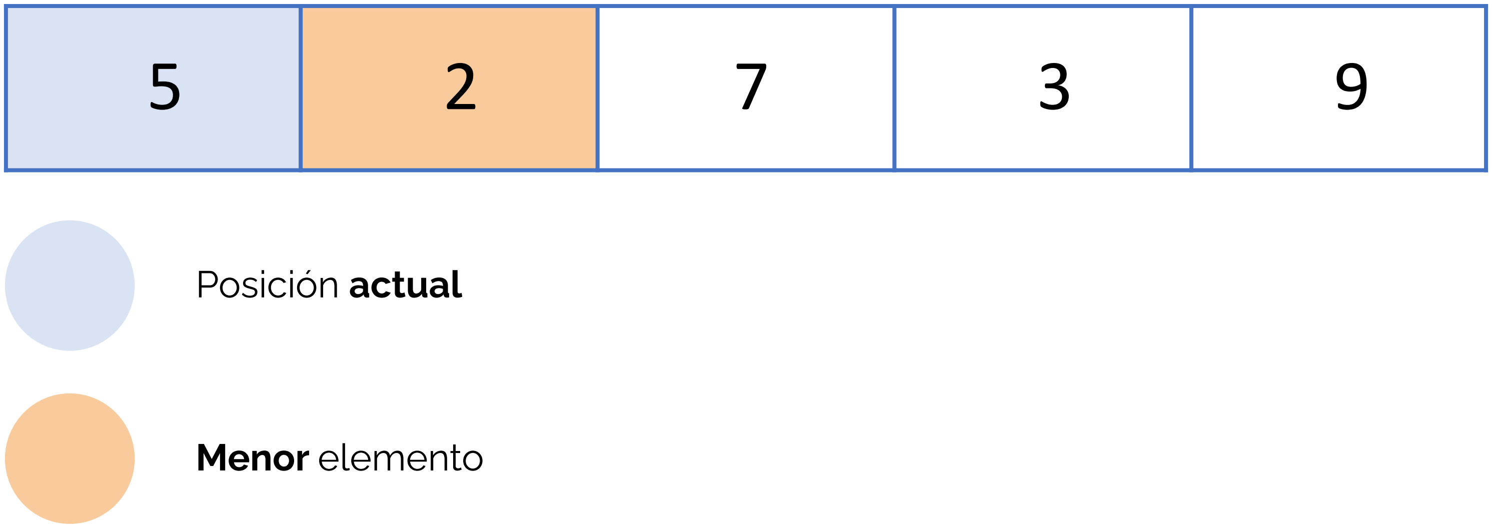
5	2	7	3	9

Nos ubicamos en el
comienzo del arreglo

5	2	7	3	9

Posición **actual**

Seleccionamos el **menor** elemento del arreglo



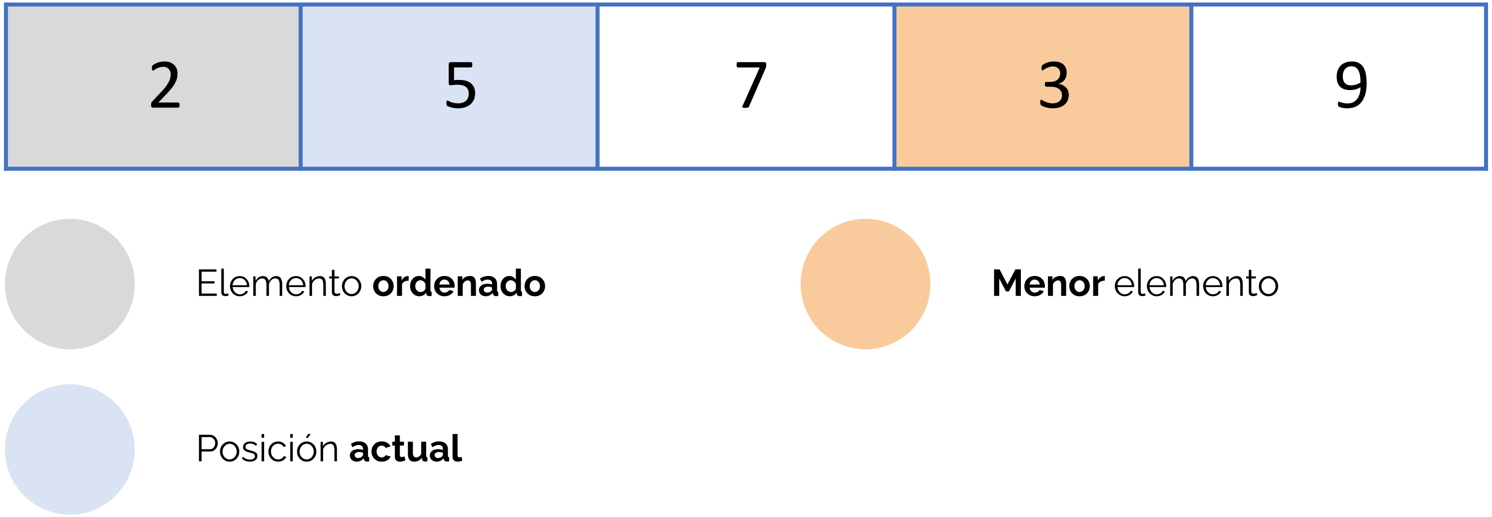
Intercambiamos el **menor** elemento con el elemento **actual**

2	5	7	3	9

Posición **actual** y **menor** elemento

Dejamos el elemento ya **ordenado** y aumentamos la posición **actual** en 1

Seleccionamos el **menor** elemento del arreglo



Intercambiamos el **menor** elemento con el elemento **actual**

2	3	7	5	9

Elemento **ordenado**

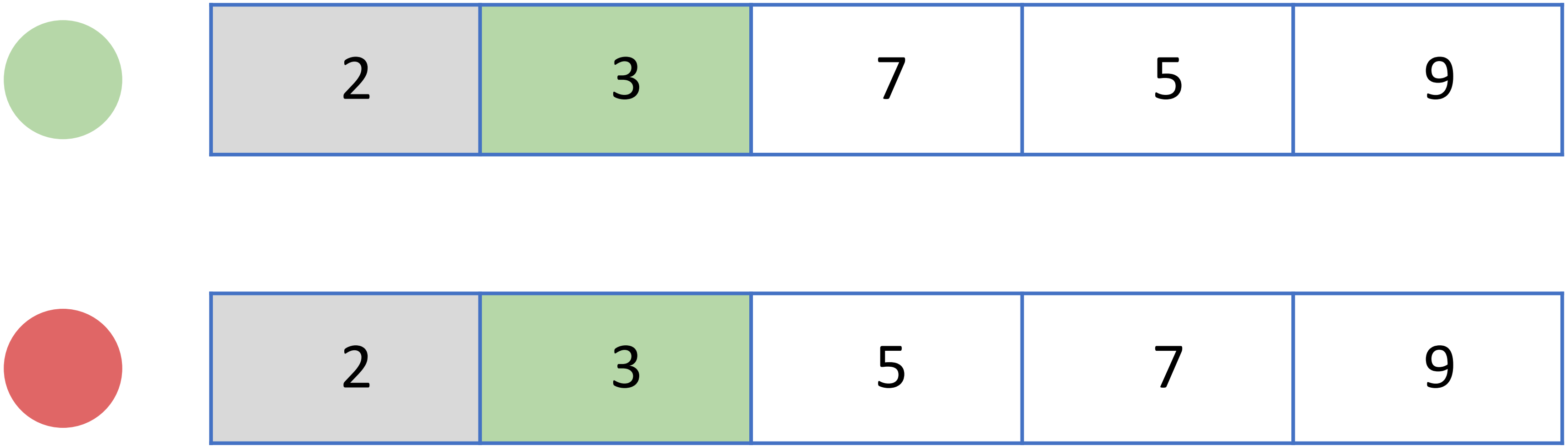
Posición **actual** y **menor** elemento

¿Por qué el arreglo se **intercambia** y no se *desliza?*

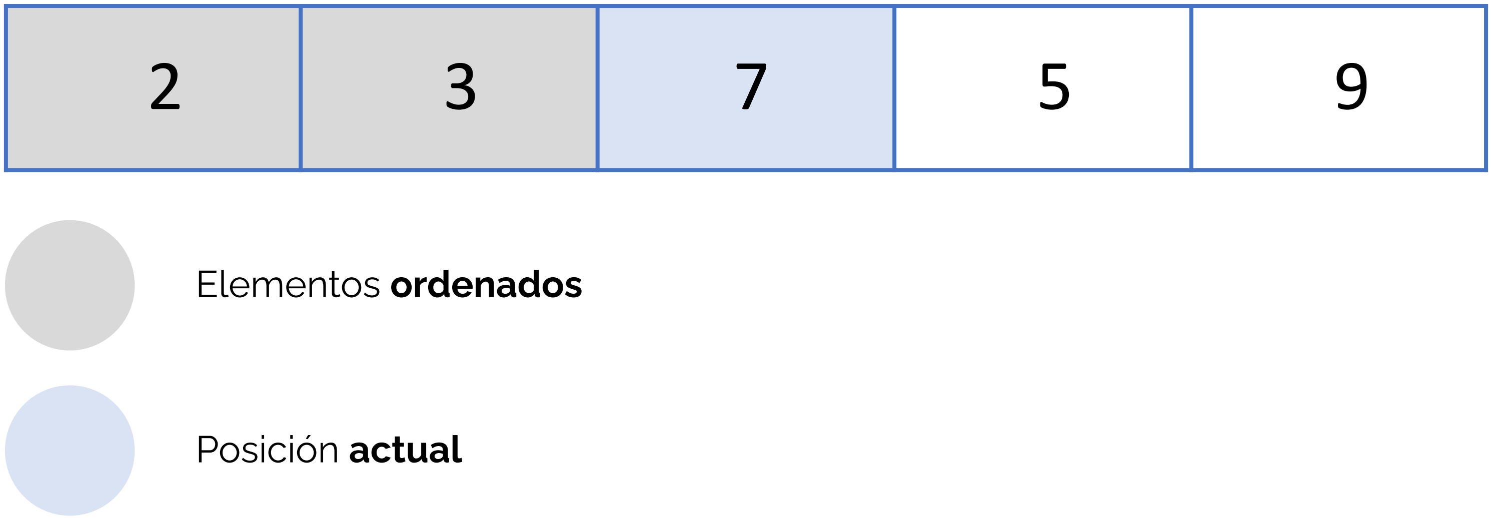
2	3	7	5	9

2	3	5	7	9

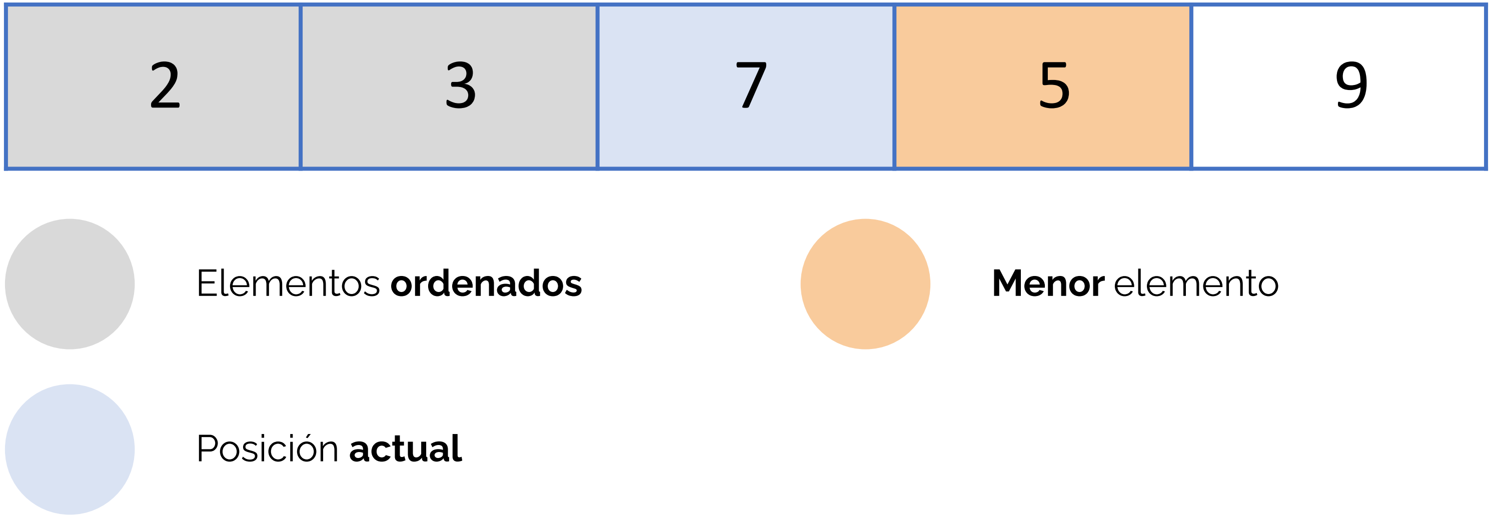
Recordemos que los arreglos son **posiciones** en memoria y no listas de Python



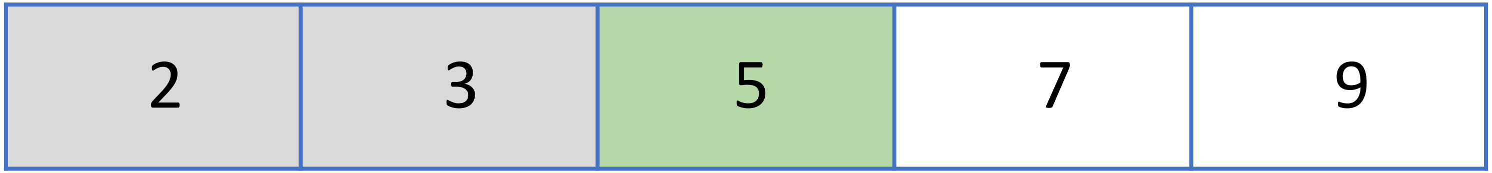
Dejamos el elemento ya **ordenado** y aumentamos la posición **actual** en 1



Seleccionamos el **menor** elemento del arreglo



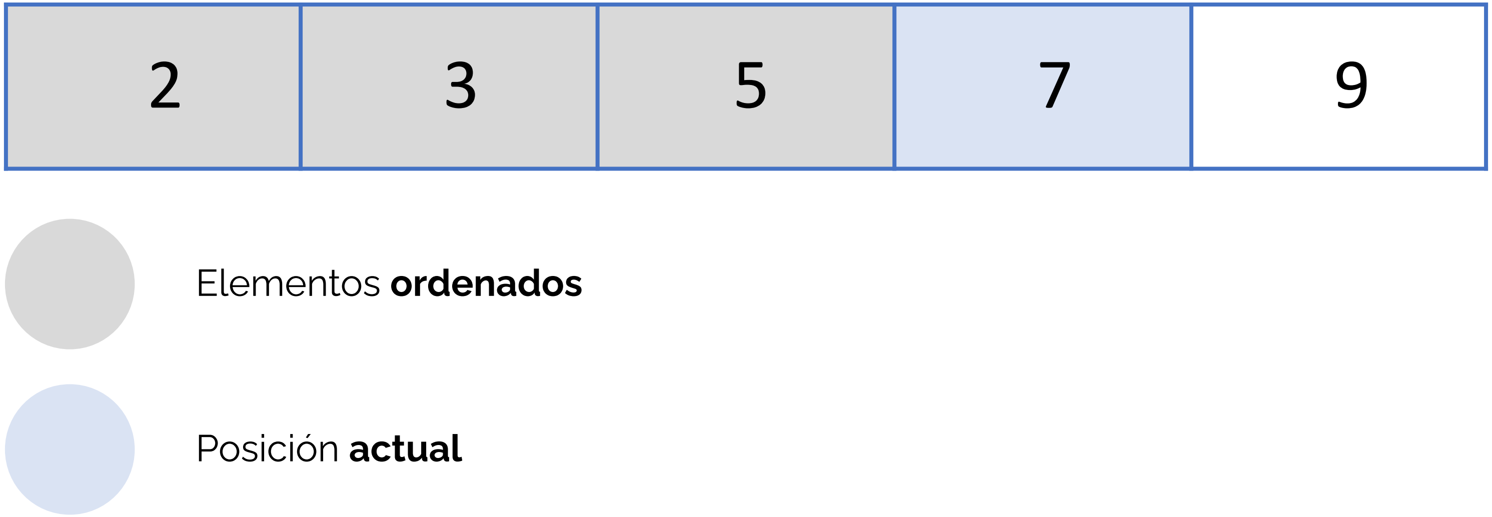
Intercambiamos el **menor** elemento con el elemento **actual**



Elementos **ordenados**

Posición **actual** y **menor** elemento

Dejamos el elemento ya **ordenado** y aumentamos la posición **actual** en 1



Seleccionamos el **menor** elemento del arreglo

Elementos **ordenados**

Posición **actual** y
menor elemento

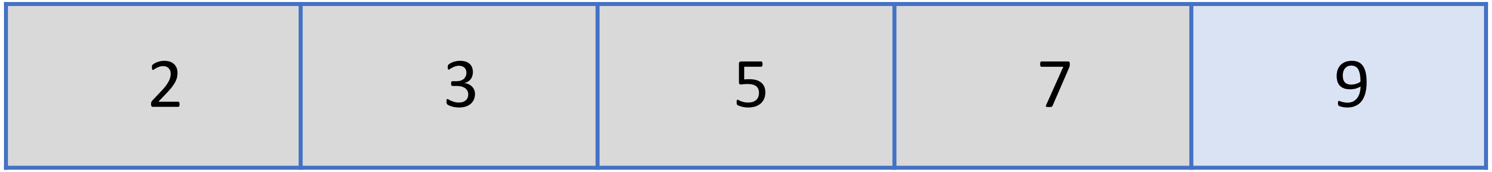
Intercambiamos el **menor** elemento con el elemento **actual**

2	3	5	7	9

Elementos **ordenados**

Posición **actual** y **menor** elemento

Dejamos el elemento ya **ordenado** y aumentamos la posición **actual** en 1



Elementos **ordenados**

Posición **actual**

Como ya es la **última** posición sabemos que el array está ordenado

2	3	5	7	9

Elementos **ordenados**

Insertion Sort

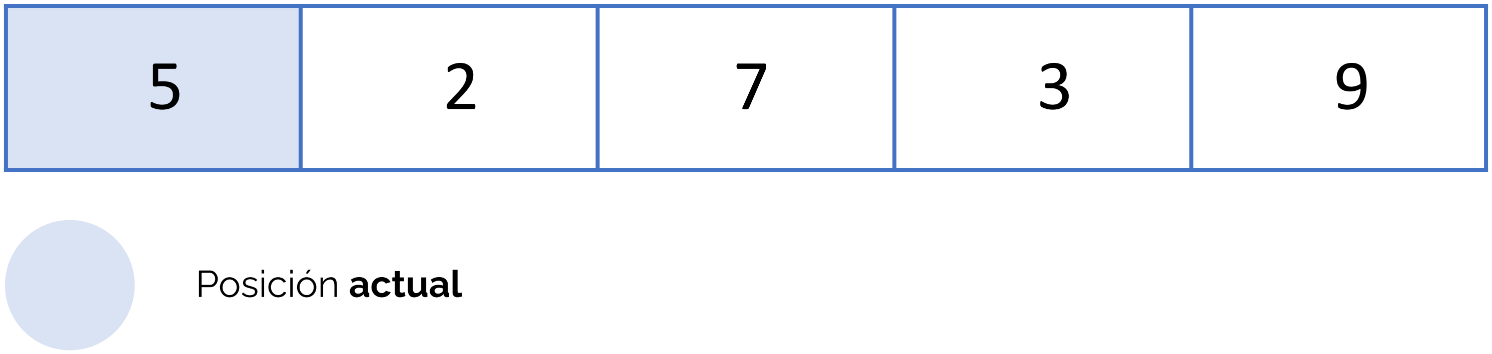
1. Tenemos una secuencia **desordenada**
2. Tomar el **primer dato** ‘x’ de la secuencia
3. **Insertar** ‘x’ en los elementos **anteriores** de manera que quede **ordenado**
4. **Avanzar uno** en la secuencia
5. **Si** quedan elementos en la secuencia, volver a 2

Insertion Sort

Veamos el mismo ejemplo anterior

5	2	7	3	9

Nos ubicamos en el **comienzo** del arreglo



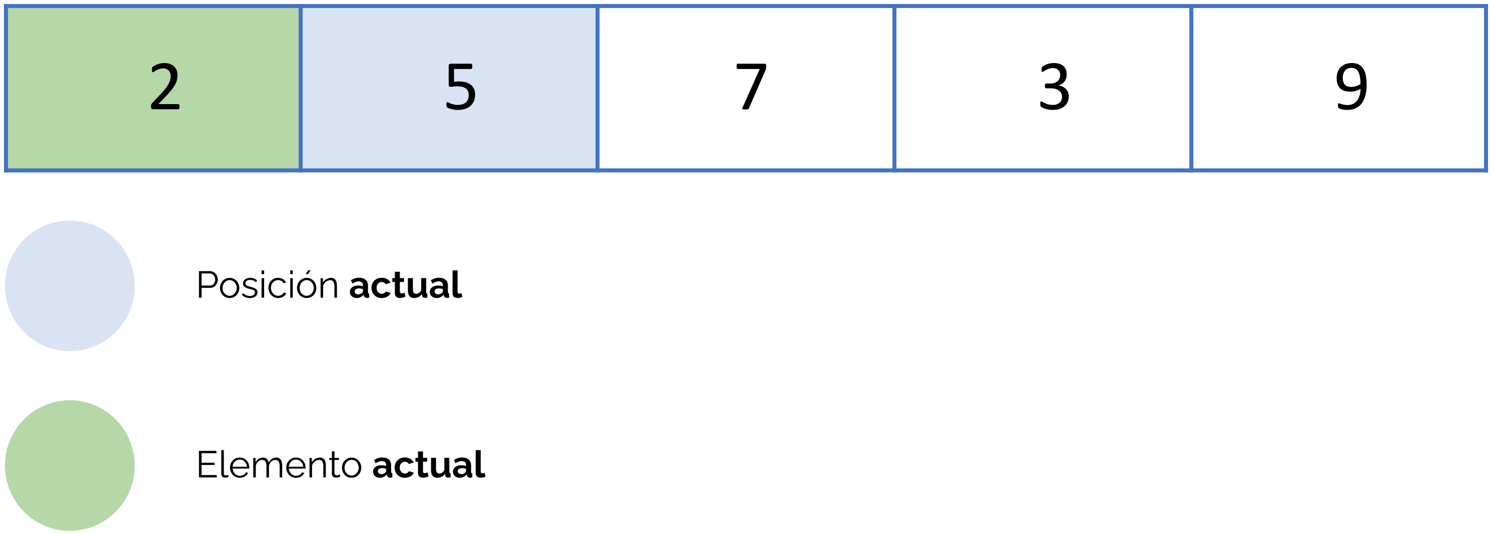
Como no hay elementos **anteriores**
dejamos tal cual y **avanzamos** una posición

5	2	7	3	9

Posición **actual**

Buscamos elementos **anteriores**

Comparamos con los elementos **anteriores** e insertamos **ordenadamente**



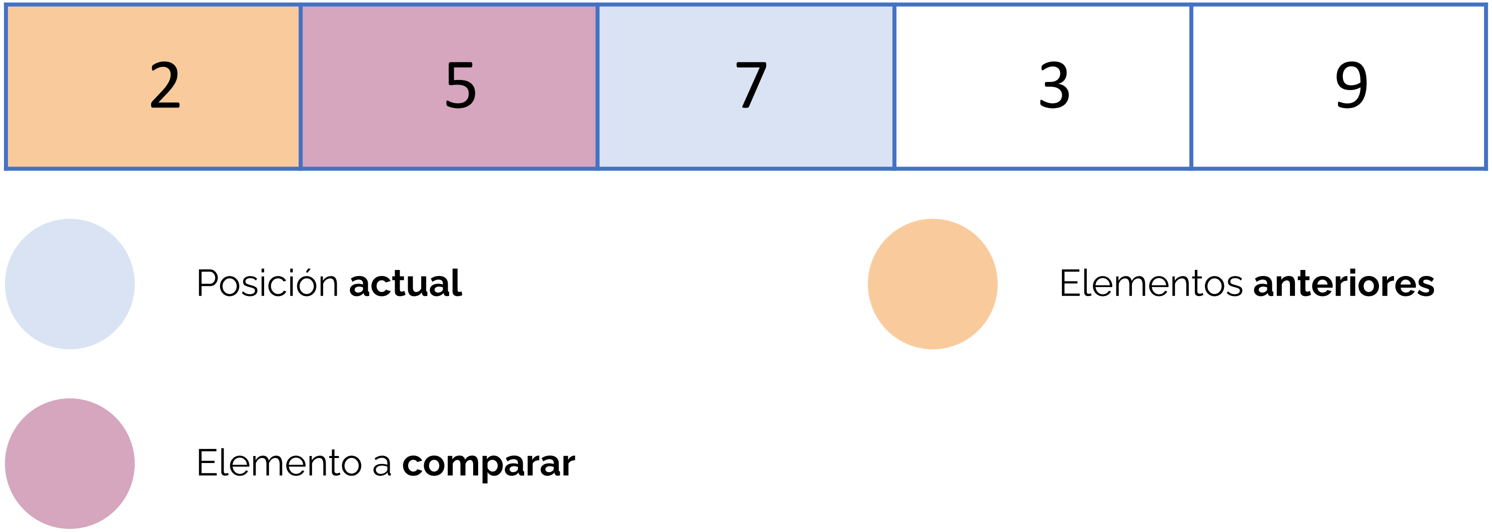
Como no quedan elementos **anteriores** dejamos tal cual y **avanzamos** una posición

2	5	7	3	9

Posición **actual**

Buscamos elementos **anteriores**

Comparamos con los elementos **anteriores** e insertamos **ordenadamente**



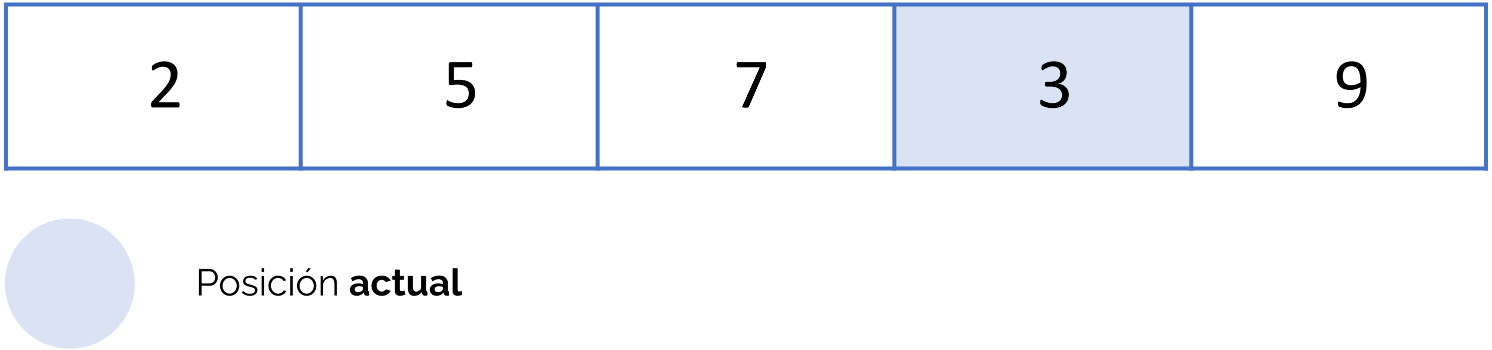
Como 7 > 5 sabemos que **no** hay elementos mayores anterior al 5

2	5	7	3	9

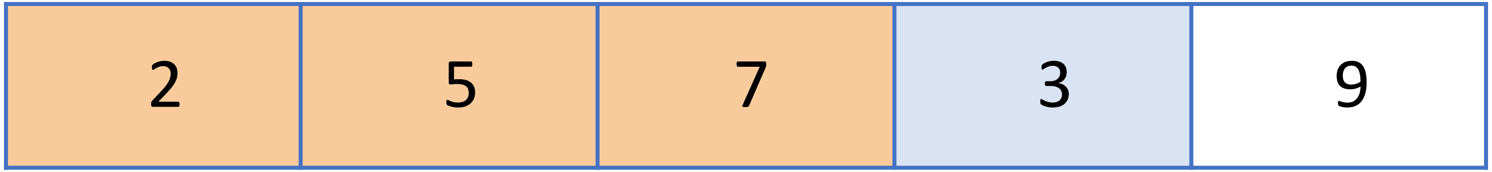
Posición **actual**

Elemento a **comparar**

Como no quedan elementos **anteriores** dejamos tal cual y **avanzamos** una posición



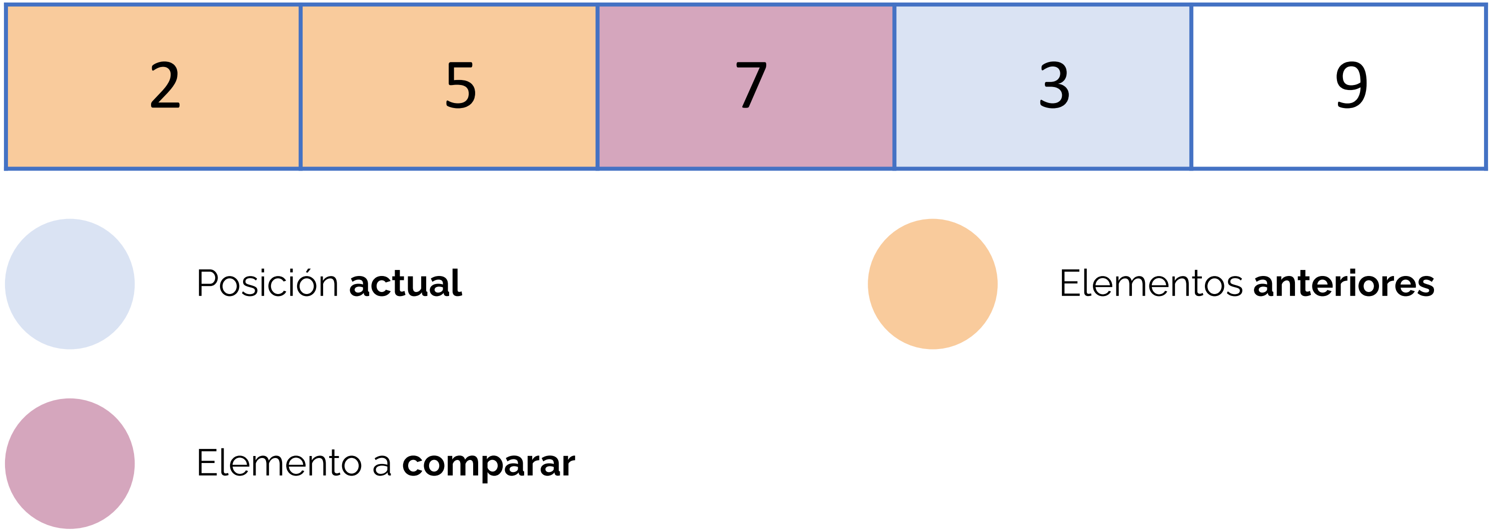
Buscamos elementos **anteriores**



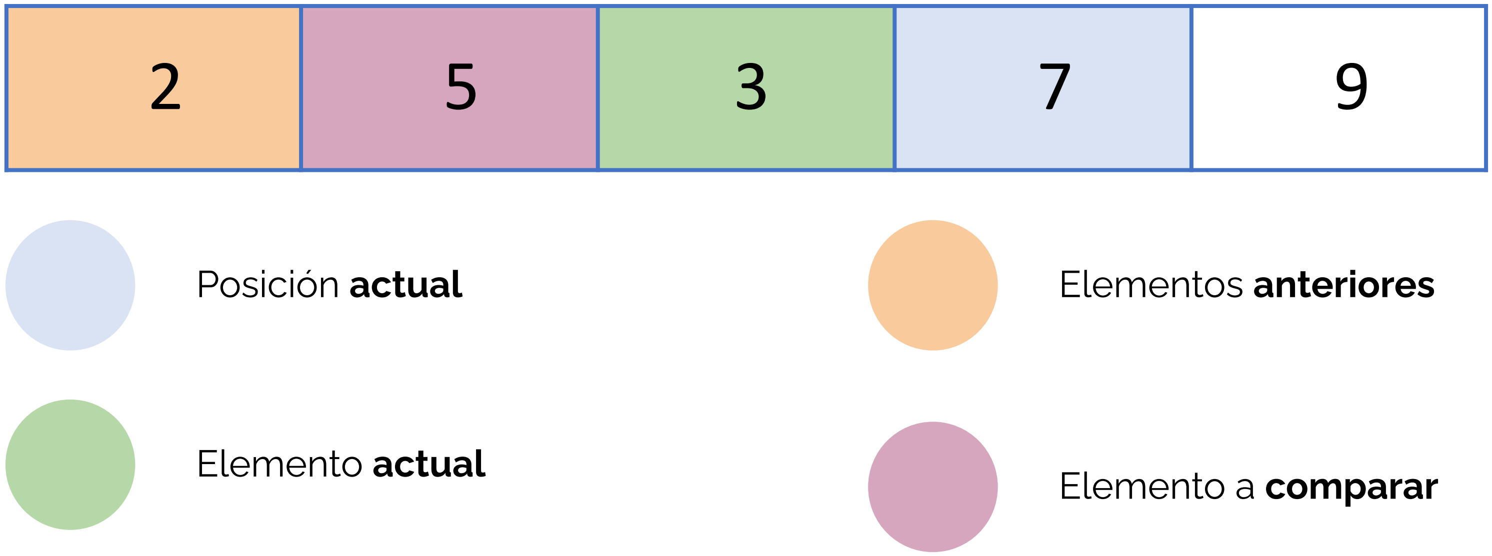
Posición **actual**

Elementos **anteriores**

Comparamos


2
5
7
3
9
Posición **actual**
Elementos **anteriores**
Elemento a **comparar**

Intercambiamos y seguimos comparando elementos **anteriores**



Intercambiamos y seguimos comparando elementos **anteriores**

2	3	5	7	9

Posición **actual**

Elemento a **comparar**

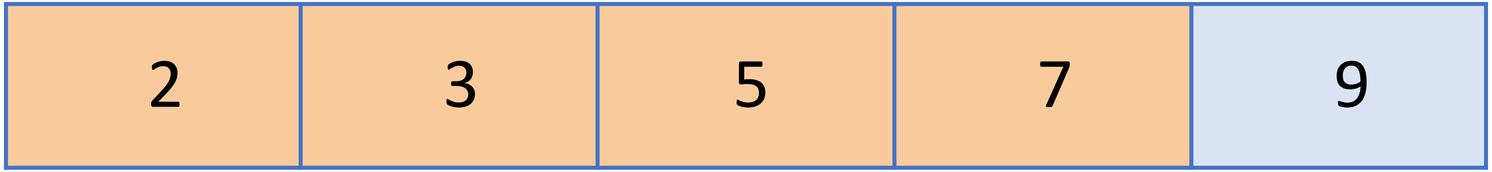
Elemento **actual**

Como no quedan elementos **anteriores** dejamos tal cual y **avanzamos** una posición

2	3	5	7	9

Posición **actual**

Buscamos elementos **anteriores**



Posición **actual**

Elementos **anteriores**

Buscamos elementos **anteriores**

2	3	5	7	9

Posición **actual**

Elemento a **comparar**

Elementos **anteriores**

Como 9 > 7 **no** intercambiamos y terminamos

2	3	5	7	9

Elementos **ordenados**

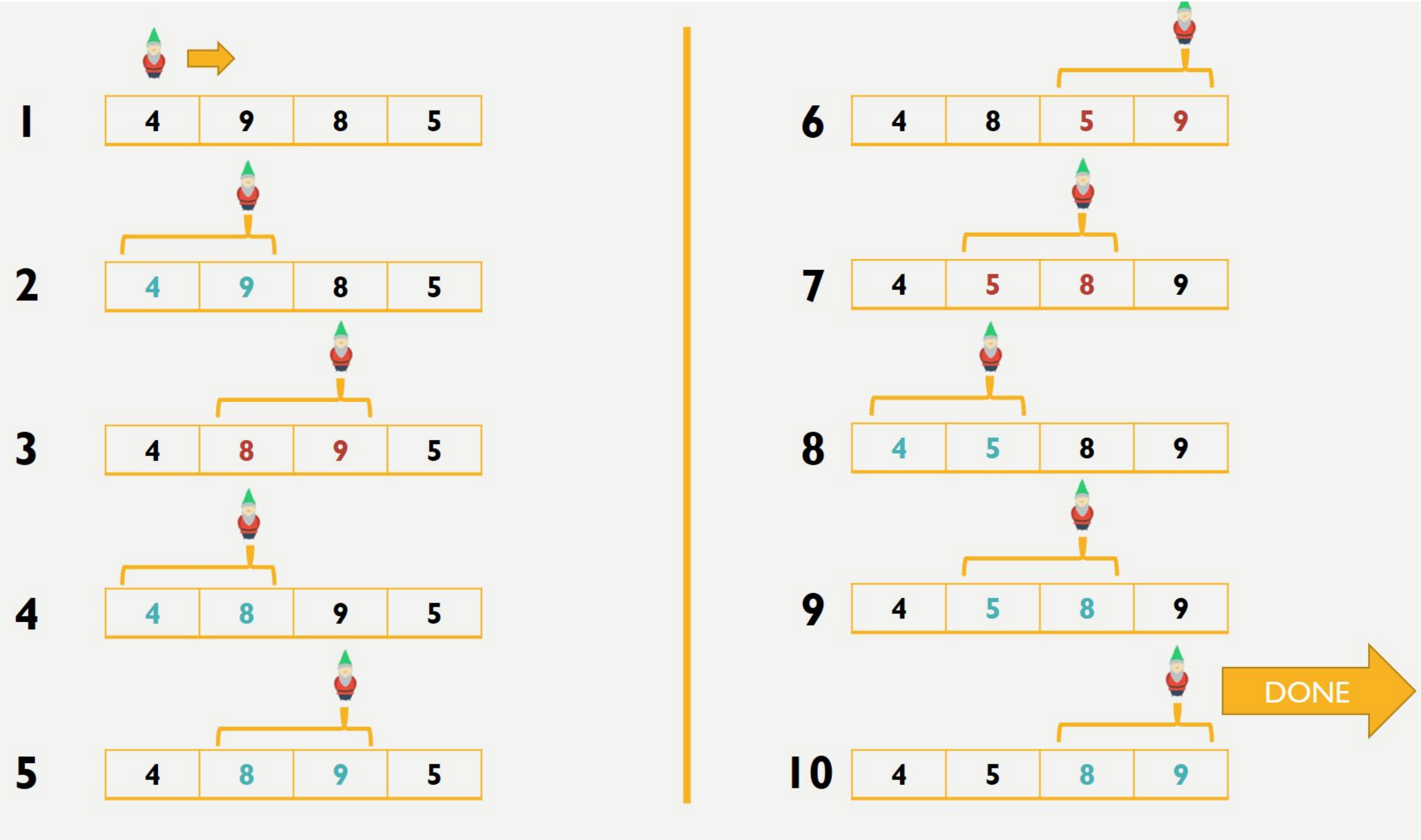
Gnome Sort

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 el puntero en 1, comparando en pares dejando cada dato ordenado.
5. Avanza nuevamente el puntero, comparando de a pares, (vuelve a 2). Si se termina la secuencia se termina el algoritmo.

Gnome Sort

Observación

GnomeSort es muy similar a **insertionSort**. La diferencia es que gnomeSort realiza ciertos pasos extras para continuar las comparaciones. Están invitados a revisar los algoritmos para entender dónde están los pasos extras.


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

Meme-sort

- **bogoSort**. Hago un reordenamiento aleatorio del array y revisamos si está ordenado, super eficiente :).
- **miracleSort**. revisamos el array y vemos si está ordenado. si no está ordenado lo volvemos a revisar y así sucesivamente. ¡Sería un milagro si funcionara!
- **gnomeSort** también se conoce como **stupidSort**.
- **Intelligent design sort**. Hay una probabilidad de 1/(n!) que el arreglo venga ordenado, solo requiere un salto de fe O(1).

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